"Extraction and Generalization of Expert Advice"

by

Paul Benjamin

Technical Report #227

June, 1986

"Extraction and Generalization of Expert Advice"

by

Paul Benjamin

Technical Report #227

June, 1986

EXTRACTION AND GENERALIZATION OF EXPERT ADVICE

D Paul Benjamin

Malcolm C. Harrison - Advisor

ABSTRACT

This work describes a method for representing knowledge in production systems which makes use of the conflict set. This permits a rich description of task situations, and allows the use of control productions to effect conflict resolution. A set of extensions to the OPS5 production system is described which facilitates the implementation of this approach within OPS5. This extended system is then used to implement a multi-level, goal-directed production system for the construction of expert systems, CAMERA, in which control information is automatically built from the actions of an expert trainer. This control information consists of sequencing and goal information which is interactively extracted from the trainer by CAMERA, and generalized by DISC, which models generalization as the process of finding 'discriminating' features, which are those features of a situation that cause a particular method to be chosen, and then constructing a description of those features. When solving a task, CAMERA examines only the discriminating features specified in the generalized control rules. Thus, instead of matching all the productions against the working memory, CAMERA considers only the relevant rules. Experiments with the system are described.

Acknowledgements

I would like to thank my advisor, Malcolm Harrison, for the advice and guidance he has provided, and for the great deal of insight that I have gained from working with him.

Also, I thank Ralph Grishman and Ernie Davis, who read drafts of this dissertation, and provided valuable criticisms.

Finally, I thank my parents, who motivated me to learn as much as I could, and put up with my seemingly eternal education. Their unflagging support and encouragement are the primary factors in my education.

Table of Contents

EXTRACTION AND GENERALIZATION OF EXPERT ADVICE

D Paul Benjamin

Malcolm C. Harrison - Advisor

1. The Problem

One of the outstanding problems in constructing 'expert' programs is that of how to extract the necessary information from a human expert. Following Lenat [Lenat83], the process of translating an expert's knowledge into operational form can be thought of as having two main steps: the statement of the trainer's knowledge in terms of 'IF-THEN' rules; and the translation of these rules into a machine-executable form.

$$\begin{tikzcd}
\text{trainers knowledge} \arrow[r, "\text{extraction}"] & \text{IF-THEN rules} \arrow[r, "\text{translation}"] & \text{executable code}
\end{tikzcd}$$

In order to model the trainer's advice effectively, the range of the extraction function should be identical to the domain of the translation function. Thus, in order to facilitate the extraction process, a translation process should be chosen which has a representation which is as close as possible to the form of the trainer's advice. In the construction of expert systems, the chosen translation process has often been a production system architecture [Bennett78, Bennett80, Bennett81, Buchanan77, Heiser78,

This work was supported in part by ONR contract number N00014-75-C-0571.

Kunz78, Reboh80, Yu79]. Thus, it has often been necessary to convert the trainer's advice into a form which is suitable for processing by a production system. It has been suggested that production systems can effectively model human problem solving [Newell73]. This work describes difficulties which the usual production system methodology can cause in the extraction and representation of human problem-solving knowledge in action domains, particularly in the selection among various methods for achieving a goal, and describes a method of representation which facilitates programming in production systems, particularly the implementation of hierarchical production systems. This representation, which describes properties of the conflict set, is then used to implement a goal-driven system for the extraction of expertise from a human expert.

A crucial problem which must be addressed by any work of this sort is that of generalization. If an interactive expert system is to be at all useful, it must be able to extend the trainer's advice to situations which are not identical to those encountered in the training sessions. The power of the generalizer dictates the ultimate usefulness of the whole system, and the power of the generalizer, in turn, is largely constrained by the representation is employs. The method of representation presented in this work describes situations in action domains by characterizing the set of possible actions.

Several decisions in the design of CAMERA were made specifically to aid the task of generalizing the control information. These decisions are discussed in Chapter 3, which is devoted to the design of CAMERA, the underlying interactive system, and their utility in generalization is discussed in Chapter 5, which describes DISC, the generalizer.

2. Expressive Power of Production Systems

In choosing a system architecture, an important decision which affects virtually all components of the system is the choice of the method of knowledge representation [Amarel68, Newell82b]. Within the production system architecture, knowledge is usually represented in two forms: the state of the task is represented in a declarative form in the working memory; and the actions which the system can execute are represented procedurally as productions. It has been suggested by Stolfo [Stolfo82] that the conflict set (the set of rules which can fire) can be thought of as a description of working memory. This work extends this notion in a systematic way, by describing a language which is built on the conflict set, which is used to represent task situations in production systems. The thesis of this paper is that the conflict set is a readily available data structure in production systems, which contains all the relevant procedural knowledge at each step in the solution of a task, and which forms a useful description of the state of the task. In particular, production system control decisions, i.e., conflict resolution decisions, should not be based upon the isolated characteristics of each production, but rather upon characteristics of the conflict set as a whole. This is in contrast to conflict-resolution strategies that have been previously suggested [McDermott76, Rychener76]. Previous conflict-resolution strategies have concentrated on such ideas as a fixed ordering of productions and the relative recency of the referenced data. Although the idea of referencing the conflict set has been mentioned [Rychener76], it has not been used as a fundamental representational mechanism, and particularly not to implement a domain-dependent conflict-resolution mechanism. One team of concurrent researchers [Laird84] employs a somewhat similar notion, which is discussed more fully in section 3.1 .

2.1. An Example

In the standard production system design [Forgy81, Newell73], a production's left-hand side may reference only working memory elements. The choice of which rule to fire is made according to an inflexible conflict resolution procedure. As a result, a programmer must embed control information in the productions in order to specify any control structure other than that provided by the conflict resolution procedure. Thus, the task of programming a production system for a particular problem requires translating the control structure of the programmer's solution into the control structure of the conflict resolution procedure.

For example, if a robot needs to decide whether or not it is safe to enter a room, it may know that it is safe to enter if there is a lit oil lamp in the room, or if there is the smell of gas, but not if both conditions are true. A standard production system (without the ability to reference the conflict set) can be used to represent this information in a straightforward way with rules along the lines of:

```
(p can-explode
   (smell ^what gas)
   (oil-lamp ^status lit ^location <x>)
   -->
   (make might-explode ^location <x>) )
```

Once this rule has fired, it can then influence the robot's action by inhibiting movement into the room:

```
(p move
   (goal ^type enter ^location <x>)
 - (might-explode ^location <x>)
   -->
   (call enter <x>) )
```

A solution of this form has several drawbacks:

[1] Special elements are needed to remember relationships. A system which is extracting this information from a human (non-computer expert) trainer would have to be able to recognize the need for these elements, understand their significance, and be able to contruct new ones, if necessary. In the above example, the token "might-explode" must be constructed.

[2] The manipulation of these markers requires the firing of additional rules. These firing cycles decrease the system's problem-solving efficiency. In the example, the rule "can-explode" must be fired in order for the rule "move" to have the proper information.

[3] The number of productions may be increased. For instance, productions may be needed just to delete or update these markers. In the example, if

the lamp goes out, the "might-explode" marker must be removed from working memory. This requires yet additional rules.

[4] An additional control structure is necessary to ensure that the rules fire in the proper order. If "can-explode" has instantiated, but has not fired, then "move" *must not* be chosen by the conflict resolver to fire. Doing so would ignore the system's knowledge. Furthermore, all rules which insert relationship tokens such as "might-explode", or which remove or update those tokens, must be fired before rules whose actions depend upon those relationships. These relationship-detecting rules may depend upon other relationships, which may depend upon other relationships, and so on. All instantiations of these rules must be fired in the correct sequence, or the system will fail to use its knowledge correctly. A self-modifying (learning) system must therefore understand this control structure and be able to modify it. This is analogous to writing a self-modifying program in FORTRAN, which would require the program to understand how to program in FORTRAN.

These drawbacks decrease the attractiveness of the production system architecture, by violating two of the most important reasons for using production systems [Pylyshyn84]:

1) There should be no hidden control apparatus. A solution's control structure should be explicitly modeled, by means of goal tokens in the working memory.

2) Production systems should be modular. It should be easy to add new rules. Furthermore, individual productions should be meaningful, rather

than needing to be seen in the context of a group of rules.

However, if the productions are allowed to access the conflict set, a solution can be obtained which avoids these drawbacks:

```
(p explode action
  (smell ^what gas)
  (oil-lamp ^status lit ^location <x>)
  →
  (call explode <x>) )

(p move action
  (goal ^type enter ^location <x>)
 - (production ^pname explode ^v1 <x>)
  →
  (call enter <x>) )
```

These rules contain no special markers to remember relationships among task elements. The rules do not require a control structure, as only *one* cycle is required to decide whether or not to enter the room. Working memory contains no marker tokens, so there are no rules needed to remove or update these tokens. The system is order-independent, since it makes no difference whether the system instantiates "explode" or "move" first; at the end of the instantiation process, an instantiation of "move" will be present if and only if the appropriate instantiation of "can-explode" is not present.

Due to the order-independent nature of the system, when an instantiation is chosen to fire after the completion of the matching process, the conflict set contains a set of instantiations which accurately reflects the state of the working memory. There are no left-over tokens describing relationships which have changed, or missing tokens which should be representing an existing condition.

Thus, conflict set access gives the system a type of "look-ahead" relative to other production systems, in that a condition which depends upon other conditions (which may in turn depend upon other conditions) need not be detected by firing productions. Such a condition instead instantiates within one cycle. This can help prevent the system from investigating inferior alternatives.

2.2. The Role of the Conflict Set in Production Systems

The previous example illustrates how a production system that accesses the conflict set can help formulate a solution which is simple and "natural" in the sense that the rules embody knowledge about the problem without interference from machinery necessary to control the production system. This suggests that the conflict set can be more than just a set from which a conflict resolver makes a choice; it can provide a description of the state of the task and of the state of the problem-solver. This information can be used to control the behavior of the system by making the conflict resolution decisions.

In any production system, the productions contain all the knowledge that the system retains. Those productions which instantiate during any given cycle reflect the relevant knowledge at that step of the task. Thus, the conflict set can be viewed as containing relationships among those task elements which are "important" in the sense that they can effect actions or enable conditions.

When an instantiation is referenced, it is as though all the elements in the instantiation's left-hand side were being referenced. Thus, the conflict set is a short-hand which can permit a brief description of necessary conditions, such as, "If you're hungry, but you can't go to the store, then go to a restaurant." The condition, "can't go to the store" may be a complex condition depending upon many factors, such as the time of day, or the day of the week, or the nearness of a store, etc.

But a rule is more than just a short-hand for a set of elements. It not only represents the elements in the rule's left-hand side, but the relationship among those elements. For example, if a production R is:

R: (A B -C → (fire A))

where A,B and C are themselves productions,

A: (X1 X2 → action1)

B: (X2 X3 → action2)

C: (X2 X4 X5 → action 3)

it is possible that some task elements may appear on the left-hand sides of two, or even all three of the productions A,B and C. R depends upon all the elements of the left-hand sides of A and B, and also upon the "relationship" C *not* holding, even though some elements of C may be present. Production R cannot be replaced by a single production, such as

(X1 X2 X3 -X4 -X5 → action1)

because this rule is not equivalent to R (it requires that *both* X4 and X5 be absent, rather than either one.) A separate rule is necessary to detect the presence of X4 and X5, and dump a token T into working memory to denote this perception. Then R could be rewritten as:

$$R2: (X1\ X2\ X3\ \text{-}T \rightarrow \text{action1})$$

However, this mechanism requires the construction of the new production to create the token T, which leads to the difficulties described in the robot example of the previous section. Even if a production system has a disjunction primitive (or predicate calculus is used as the representation) so that it can express R2 as:

$$R3: (X1\ \&\ X2\ \&\ X3\ \&\ (\text{-}X4\ |\ \text{-}X5) \rightarrow \text{action1})$$

R3 still lacks the knowledge content of R. For example, X2 occurs implicitly three times in rule R, once in each production (A,B,C) that is referenced. X2 may perform a different role in each referenced production. In rule A, for instance, X2 might be a box which supports another box. In rule B, X2 might be a box which contains something. In rule C, X2 might be a box which is on top of another box. At some future time we may wish to apply R to a new situation in which X2 may correspond to two or three different boxes which perform these separate roles. This one-to-many correspondence constitutes one of the more difficult aspects of generalization, as it precludes viewing generalization as an isomorphic process [Hayes-Roth78]. R is more appropriate for such generalization than R2 because R uses the conflict set representation to embody the structure of the relationships among the condition elements, and thus provides a richer description of the situation than a representation based solely upon working memory elements.

A second way in which the conflict set can be used in production systems is to use domain-dependent rules to effect the conflict resolution. For example, in the monkey-and-banana task, if A is the rule "put a box on top of the box", B is the rule "crawl onto the box", and C is the rule "the banana is within arm's length from the box", then the **control** rule:

$$(A\ B\ C\ \rightarrow (\text{fire } B))$$

would choose to crawl onto the box, rather than increasing the height of the pile of boxes, since the banana can be reached. This is not equivalent to a fixed, partial ordering of the productions. A fixed ordering could express a precedence such as:

$$(A\ B \rightarrow (\text{fire } A))$$

but would fail to represent situations in which the precedence between two rules is dependent on factors other than the two rules. For example, the two **control** rules:

$$(A\ B\ {-C} \rightarrow (\text{fire } A))$$

$$(A\ B\ C\ \rightarrow (\text{fire } B))$$

might exist in the above example. A fixed precedence ordering between A and B is inadequate to achieve the same effect as these two **control** rules. Using conflict set access in this fashion implements the idea that what a system should do is a function of what it can do. The exploration of the power and limitations of this idea constitute one of the fundamental ideas of this work.

It is possible to effect this same result with production systems which cannot access the conflict set, by writing rules that detect relationships and create working memory elements to reflect them. For example, such a version of rule C above could place a token in working memory indicating that the banana is reachable from the box. This token could then be used by other rules to determine whether to climb on the box or not. However, as noted above, this mechanism not only requires additional working memory elements and extra productions, but requires a control structure to insure that all the "relationship-detecting" rules fire before any rules that depend on those relationships. This control structure must be designed into the productions, and if the system is to be able to effectively modify itself, it must understand this control structure, and be able to translate the trainer's advice into it. Thus, programming in production systems effectively amounts to translating an algorithm into the control structure imposed by the production system's conflict-resolver. The use of the conflict set representation decreases the need for this control structure by eliminating the standard, fixed conflict resolution process and instead controlling the production system with productions.

It is worth noting that early pioneers in production systems [Newell73] sometimes referred to working memory as "short-term" memory and to production memory as "long-term" memory, thus attempting to draw a parallel between these parts of production systems and human short-term and long-term memory. In its simplest form, the working memory was conceived as containing the system's perceived form of the external task environment, and the productions encoded the procedural knowledge of the system. By allowing the system to access the conflict set, it can avoid having to use working memory as a blackboard to control the productions, and can encode the control information as productions, rather than as tokens manipulated by productions.

The problem with this approach to production system architecture is that by allowing rules to access the conflict set, a new architecture is created which is not actually a production system, so that the new system loses some of the characteristics which have attracted so many researchers to production systems — specifically the data-directed nature. To remedy this problem, the use of conflict-set reference is restricted in the following way: this mechanism is implemented as an extension to OPS5, not a modification, thus allowing the use of either conflict set reference or data-directed operation, and CAMERA restricts the use of conflict set reference to the construction of production hierarchies, in which conflict resolution of a group of rules is handled by productions in a group which is higher in the hierarchy. Within a group, rules function in a data-directed fashion. Thus, CAMERA uses conflict set access to construct a "layered" production system, in which one production system can control the actions of another production system. Conflict set access is restricted to situations in which rules of a higher production set perform the conflict resolution of a lower set of production rules. CAMERA represents a mixture of data-directed control and conventional top-down control.

2.3. Usefulness of Rich Domain Descriptions

In solving a task, multiple layers of meaning may be necessary to understand the role of an object. A correct understanding of objects' roles can aid in generalization by determining which objects have importance. The conflict set representation is capable of expressing these layers of meaning. For example, if the task is driving a car, a standard production system may contain a rule resembling:

```
(p proceed-through-the-intersection action
   (light ^color green ...)
 - (in-front-of ^object person ^where car)
   ...
   →
   (call start-forward) )
```

which states that the car may proceed through an intersection if, among other conditions, a person is not in front of the car. This **action** rule is typical of the rules in production systems. However, an important layer of understanding is missing. The system does not know why a person should not be in front of the car. In order to reply intelligently to queries about this, or to understand stories about driving cars, or to generalize knowledge about driving cars to other tasks, such as sailing boats, the system needs to know why this condition element is in the rule. A system which can access the conflict set can provide this information by containing the rule:

```
(p run-over-the-object action
  (moving ^what <v1> ^direction forward)
  (in-front-of ^object <v2> ^where <v1>)
  →
  (call start-forward) )
```

and replacing the previous rule by:

```
(p proceed-through-the-intersection action
  (light ^color green ...)
 - (production ^pname run-over-the-object ^v1 car ^v2 person)
  ...
  →
  (call start-forward) )
```

This rule captures the meaning behind the requirement that no person be in the intersection. Furthermore, this process can be applied to the other condition element shown on the other rule's left-hand side. What is the significance of the green light? Rather than putting a data structure in working memory with information about traffic lights, another rule could be added:

```
(p move-on-green-light action
  (light ^color green)
  →
  (call start-forward) )
```

and reference to this rule could replace the reference to the traffic light in the previous rule. The resulting rule is:

```
(p proceed-through-the-intersection action
   (production ^pname move-on-green-light ...)
 - (production ^pname run-over-the-object ^v1 car ^v2 person)
   ...
 -->
   (call start-forward) )
```

This process would be continued until this rule referenced only other rules. In this way, the system references objects only through their roles in effecting actions or enabling conditions. The result is a frame-like hierarchical representation, which groups objects together functionally, and in which rules of any group access rules in lower groups.

Extending a production system to access instantiations in the conflict set proves to be a useful programming tool. This idea has been implemented in the CAMERA system, which is discussed in detail in Chapters 3 and 4. As this is an extension, not an alteration, of the production system architecture, it permits the programmer to write rules either with or without conflict set access, and allows existing rule sets (without conflict set access) to execute. Furthermore, this allows the programmer to create a hierarchical structure of production sets, in which each production set controls the conflict resolution of the production sets below it. For this reason, this architecture has proved particularly useful in the modeling of multi-level control structures.

2.4. An Example of the Use of the Conflict Set

The following brief example illustrates how references to the conflict set can be used in a production system.

Sleeman [Sleeman83] investigates the induction of models of students' skill in algebra. He provides a set of ten production rules which are sufficient to solve the linear algebraic equations which constituted the task domain. These ten rules, written in their original stylized format for simplicity, are:

```
(p FIN2
  (SHD X = M/N) → (SHD (evaluate M/N)) )

(p SOLVE
  (SHD M * X = N) → (SHD X = N/M))

(p MULT
  (lhs M * N rhs) → (lhs (evaluate M*N) rhs) )

(p ADDSUB
  (lhs M ± N rhs) → (lhs (evaluate M ± N) rhs) )

(p XADDSUB
  (lhs M*X ± N*X rhs) → (lhs < M ± N > * X rhs) )

(p NTORHS
  (lhs ± M = rhs) → (lhs = rhs ∓ M) )
```

(p REARRANGE
  (lhs $\pm$ M $\pm$ N*X rhs) $\rightarrow$ (lhs $\pm$ N*X $\pm$ M rhs))

(p XTOLHS
  (lhs = $\pm$ M*X rhs) $\rightarrow$ (lhs $\mp$ M*X = rhs))

(p BRA1
  (lhs < N > rhs) $\rightarrow$ (lhs N rhs))

(p BRA2
  (lhs M* < N*X $\pm$ P > rhs) $\rightarrow$ (lhs M*N*X $\pm$ M*P rhs))

where M, N and P are integers; lhs, rhs, etc. are general patterns; $\pm$ denotes plus or minus; $\mp$ denotes minus or plus; < and > represent algebraic parentheses; "evaluate" signifies evaluation of the algebraic quantity; and SHD designates the string head. These rules are linearly ordered, i.e., conflicts are resolved by choosing the instantiated rule which occurs closest to the beginning of the rule set. These rules implement an algorithm for solving linear equations. Note that an error in the rule ordering in [Sleeman83] has been corrected.

The conflict set representation can be used to represent this algorithm with the following rules:

(p DIST action
  (lhs M * < N*X $\pm$ P > rhs) $\rightarrow$ (lhs M*N*X $\pm$ M*P rhs))

```
(p TORHS action
 - (instantiation of DIST)
 - (instantiation of TERM-CONTAINS-X (with t = term))
   (lhs ± term ± lhs2 = rhs) → (lhs ± lhs2 = rhs ∓ term) )

(p TOLHS action
 - (instantiation of DIST)
   (instantiation of TERM-CONTAINS-X (with t = term))
   (lhs = rhs ± term ± rhs2) → (lhs ∓ term = rhs ± rhs2) )

(p COLL action
   (lhs M*X ± N*X rhs) →
       (lhs < M ± N > * X rhs) )

(p EVAL action
 - (instantiation of DIST)
 - (instantiation of TORHS)
 - (instantiation of TOLHS)
 - (instantiation of COLL)
   (lhs M ±|*|/ N rhs) → (lhs (evaluate M ±|*|/ N) rhs) )

(p SOLVE action
   (SHD M * X = N) → (SHD X = N/M) )

(p BRACE action
   (lhs < M > rhs) → (lhs M rhs) )
```

```
(p TERM-CONTAINS-X sensing
   (lhs ± t ± rhs)
  - (lhs ± t1 ± t2 ± rhs)
   (lhs ± t3 X t4 ± rhs) → )
```

This set of rules contrasts in several ways with Sleeman's original system of rules:

1) There are fewer rules than in the original system. Altogether, the new system contains eight rules, instead of the original ten.
2) The concept of a term, and of whether or not it contains the variable, is explicitly present in a **sensing** rule, rather than unstated, as in the original rules. This knowledge is represented separately from the actions.
3) The control embodied in these rules is not the same as Sleeman's original system, which constituted a fixed algorithm which, given a particular linear equation as input, would always pursue the same steps in its solution. The new rules are not linearly ordered, as the original rules are. Dependencies among the new rules are explicit. In the new set of rules, the rules DIST, COLL, SOLVE, and BRACE are coroutines, which are of equal priority. These rules can be allowed to fire in any order. Thus, for a particular linear equation, this system could pursue different paths to the solution.

3. Using the Conflict Set for Knowledge Representation in Expert Systems

To illustrate the use of the conflict set in expert system design, as both a descriptive mechanism and a conflict-resolution mechanism, a system, CAMERA (for Control And MEthod Rule Acquisition), has been implemented for the extraction of control knowledge from a human expert trainer [Benjamin83].

Kowalski [Kowalski79] views this information as having two components, a logic component and a control component. For computer design, for example, the first would include the axioms and theorems of boolean algebra and the specifications of the available chips, while the second would include the design techniques contained in a textbook on logical design. This categorization differs from the more traditional declarative/procedural distinction, since the logic component is best viewed as a declarative specification of procedural information. In general, it seems that the problem of dealing with logical information is likely to be more tractable than that of dealing with control information. Accordingly, CAMERA concentrates on the problem of extracting from the human the essential components of his control expertise.

Actually, in the tasks which were programmed in this work, I separated the control knowledge in a slightly different manner than Kowalski. The logic component in Kowalski's system contains the possible actions, together with their necessary preconditions, and his control component contains heuristic and algorithmic information about the advisability of applying particular pieces of logic information. In this work, however, logic information contains only the possible methods and actions, with no restrictions as to legal applicability, and control information contains all the restrictions. Thus, the control information makes no distinction between preconditions which are structurally necessary and heuristic restrictions on the advisability of application. This decision was made to simplify the programming of the underlying action system, and to investigate the extent to which restrictional information can be automatically acquired.

To reflect this logic/control distinction, two types of rules exist in CAMERA: those that reference task objects, and those that reference conflict set elements. The task productions, which are referred to as **action** and **method** productions in this work, contain all manipulations of task elements and goals, respectivley, and perceptual conditions which can exist in the task. The rules which can access the conflict set, called **control** rules in this work, are restricted to operate *only* in the conflict set space. This permits the construction of a "layered" production system, in which a "higher" production system can affect the actions of a "lower" production system, by accessing the conflict set of the lower system, and performing the conflict resolution. CAMERA models a two-level production system: the lower production system contains all the perceptual and goal information about the task, and rules for manipulating this information; and the upper production system contains rules for controlling the task. The lower production system's conflict set is the working memory for the upper system, as the elements of the conflict set are accessible as tokens by the control rules in the upper system.

In previous work on this problem [Stolfo79,Stolfo79b,Stolfo79c], control information was represented as descriptions of execution traces which had been generated by the expert. These were used to constrain the search space in a way similar to that discussed by Georgeff [Georgeff82,Georgeff79]. With this technique, descriptions were constructed which reflected surprisingly accurately the structure of the solution generated by the expert. However, these descriptions were in some cases too rigid, and did not degrade gracefully when the situation was not identical to previously encountered situations. In particular, the program had no notion of similarity, and was unable to recognize parts of the description which were really instances of a more general procedure. In order to improve this aspect of the system, a more powerful description mechanism was needed. Since the computational problems of analyzing sequences are extreme (at least NP-hard in general) even in the simpler non-probabilistic case, it is necessary to make use of more information about the sequences. The two sources of additional information are the production rules themselves, and the human expert. This system concentrates on extracting information from the expert, particularly about the planning implicit in his solution.

This work is somewhat similar in essence to Winston's work [Winston75], in that models are constructed and refined according to examples. However, in Winston's case, the set of examples could be chosen to aid the model construction process. In the present case, the domain under examination is goal structure rather than physical structure, and is larger and therefore more sparsely populated. This makes the sequence of examples unpredictable and possibly unrelated, and makes anticipation of the changes to the model more difficult.

3.1. Representation and Selection of Problem-solving Methods

For the purpose of constructing an expert system, or any program, which interacts with people, it has often proven advantageous to use information about the goals which people use in solving a problem [Newell82b, Schank77]. Goal-directed systems usually solve problems by back-chaining, i.e., proceeding from the desired solution backwards to the initial state, as opposed to the usual forward-chaining, data-directed way in which production systems usually operate. In CAMERA, this knowledge is represented as a hierarchical goal tree, together with **method** productions which embody possible subgoal expansions of goal nodes. Goals have a "type", and, optionally, parameters that specify details. For example, a goal may have type "find" and parameters "what = piece" and "color = red". The number of possible goal types is not fixed, as they are extracted from the trainer's input. The mechanism for goal and method extraction is explained in sections 3.2.3 and 4.4 .

This sort of goal representation differs from that adopted in [Laird83], in which goals do not have a syntactic goal "type", but instead are defined purely semantically, as the difference between the states of the problem-solver before and after the subgoal expansion. In this work, goals have a specific "type", which is extracted from the trainer. This choice is due to two causes: firstly, this system is designed to extract information from a human trainer, which that of [Laird83, Laird84] is not; and secondly, this facilitates generalization of the control information, by allowing control information to be classified according to the method chosen to satisfy a particular type of goal. This is discussed in detail in Chapter 5.

A major type of decision made by a goal-directed system while solving a problem is which method to choose at each step of the solution process. In CAMERA, this corresponds to the conflict resolution among the instantiated method rules. As pointed out above, the use of the conflict set representation permits the use of domain-dependent rules to effect conflict resolution. These conflict resolution rules, called **control** rules in this work, correspond to the "metarules" of [Davis80a, Davis80b]. The use of a multi-level rule structure, incorporating metarules and object-level rules, differs from the structure used in [Laird84], in which the control architecture merges the various levels into one level, so that metadecisions are treated in the same way as object decisions. However, [Laird84] has several different "spaces", such as a selection space, an evaluation space, and a task space, which correspond closely to the various levels of productions used in CAMERA, and these spaces are ordered hierarchically, e.g., goal selection occurs before problem space selection. CAMERA uses the multi-level control structure of metarules, with the control information implemented as metarules which choose among the method rules, in order to aid the learning process by permitting the control information and method information to be acquired independently. These two types of information are constructed in two different ways: the method information by being told by the trainer; and the control information by observation of a human trainer.

These control rules permit CAMERA to function in a different manner from that normally used by production systems, by selectively matching productions against working memory. OPS5, and most other production systems, work in a bottom-up fashion. They first match their rule set against working memory, and then perform a conflict-resolution procedure to choose an instantiation to fire. If the set of instantiations is extremely large, and each cycle can insert and remove many instantiations, then such a production system can consume a great deal of time and effort in each cycle. Fixed partitioning of the production set can reduce this expense by eliminating many rules from consideration, but this inflexible approach can separate rules that may be relevant in an unforeseen situation. Also, the partitioning process requires a mechanism that would have to be automated in a learning program. This introduces additional time and complexity, since the system would need to be able to construct, merge and separate partitions.

Selectively matching productions against working memory helps to avoid wasting time considering unnecessary alternatives. In each cycle of CAMERA, the system matches the method rules against the goal tree in working memory to determine which methods are possible. Then the control rules are searched to find those which choose among the instantiated method rules. Each control rule contains a set of conditions on its left-hand side which must be true for that rule to fire, i.e., for the method rule on the right-hand side of the control rule to be selected. CAMERA checks only these conditions to determine which control rule to fire. Since control rules access only the conflict set, these conditions are represented as instantiations of rules which must be present in the conflict set. This means that only the rules on the left-hand side of a control rule are matched against working memory. When a control rule is found whose conditions are matched, it is fired, choosing a method rule which expands a node in the goal tree. There will be one or more new active goals, and the entire process is repeated. This method of operation can reduce the expense of the matching process, as only those rules which affect the method selection need to be examined, and the entire conflict set need not be constructed.

For example, when solving a jigsaw puzzle, if CAMERA needs to find a square red puzzle piece, all CAMERA's productions concerning determination of textures and weights of pieces, as well as any other information, such as the cost of buying the puzzle, is not relevant to the decision of how to find a square red piece, and need not be matched against the current situation. The only method rules CAMERA will instantiate are those that expand a goal of type "find", and thus the only control rules which can possibly instantiate are those which choose among methods of expanding a "find" goal. The only productions which can have any effect on this instantiating process are those which occur on the left-hand sides of these control rules, so that CAMERA needs to check only the working memory elements (or, recursively, those of other instantiations) which are referenced by these productions. These working memory elements are matched according to the usual OPS5 matching algorithm [Forgy81].

It is worth noting that allowing the system to respond selectively to environmental properties in this fashion is often cited as a property of cognitive systems [Pylyshyn84].

3.2. The Design of CAMERA

3.2.1. System Components

In order to use not only detailed information about production sequences, but also planning information, the system must provide some framework for specifying such information. In the system described below, there is knowledge of the following kinds:

[1] WM elements which describe the **task domain**, and productions which permit the actions which the system must be able to perform on these domain elements. We refer to this component as the **action** system. The action productions are a set of low-level domain manipulations which are complete in the sense that all possible actions are present. Also, these action productions contain no control information whatsoever. The firing of these productions is controlled by the method and control productions, which are described below. The system provides for the augmentation of the action production set by a trainer if a desired action is not present.

[2] Productions which have an empty right-hand side. These **sensing** productions serve to detect complex conditions which are used by the system to control loops and select other rules for firing. These

productions are instrumental to the use of the conflict set representation, since they are used to describe the task domain and the goal tree. An alternative implementation of this capability would be to implement such conditions as functions accessible by the left-hand sides of productions. However, implementing such conditions as productions allows easy expansion of the descriptive capability of the system without having to resort to modification of OPS5 code.

[3] WM elements and productions which describe the goal structure used by the system. These include domain-independent **tree** productions, such as standard goal-tree operations for propagating solved or unsolvable markers; and also domain-dependent **method** productions, which describe how goals are expanded or solved. The goal tree is stored in WM because it is the trainer's representation of the task. Without the goal tree, WM contains only task objects. The goal to be accomplished is an integral part of the task, and thus is kept in WM, along with its subgoal expansion. Method rules are of the form:

$$(\text{goal} \rightarrow \text{subgoal-tree})$$

These rules contain only the subgoal expansion, without any reference to conditions under which that expansion is to be preferred. Thus, method rules constitute the context-free component of the control of the system. The control rules specify the contextual information.

[4] WM elements which describe the contents of the conflict set, and **control** productions, which use these WM elements to specify the choice of which action or method production to execute. A control rule is a

production whose left-hand side contains instantiations from the conflict set, which are represented by their working memory images, and whose right-hand side specifies one of those rules to be chosen for firing during the next OPS5 cycle. For instance, if a chess-playing production system has two method rules which reflect the two methods of castling, to the kingside and to the queenside, a simple control rule might state that kingside castling is always preferable:

```
(p choose-to-castle-kingside control
   (production ^pname castle-kingside ^type method)
   (production ^pname castle-queenside ^type method)
   →
   (call fire-production 1))
```

[5] **Top-level** productions, which control the overall execution of the system. These rules decide which of the other types of rules should be fired in any cycle. As such, these rules constitute the actual OPS5 production system. All other types of rules are manipulated, either directly or indirectly, by these rules.

[6] **Interface** and **Training** rules which implement the interface between the system and the human trainer.

For example, in the 'blocks' world, the action system would consist of the blocks themselves, together with all manipulations of blocks - picking one up, dropping it, looking at one, sensing its color, etc. The goal system might contain the goal 'build an arch', and productions to continually expand this goal until it can be described as a regular expression of action system productions. The control productions control this expansion process, choosing the most appropriate subgoal expansion. The top-level rules can choose to fire any tree productions in order to update the goal tree, then fire a control rule, which chooses a method rule to fire.

3.2.2. Task Design

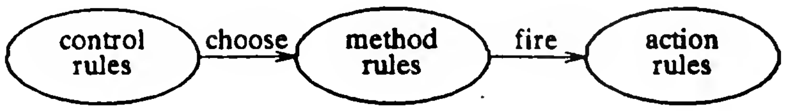
Within CAMERA, tasks are structured in the following manner in order to take advantage of the access to the conflict set.

A task's action productions manipulate the domain, and cannot access the goal tree. In the modified OPS5, an action production is not allowed to fire by itself, but instead must be activated by the presence in the goal tree of an active goal whose type is identical to the action production's name.

The domain-dependent productions that specify possible expansions of the goal tree, and the productions that choose among these subgoal expansions, are implemented as **method** productions and **control** productions, respectively. Control and method productions access the conflict set. Thus, methods can be chosen depending on which actions are fireable, which task conditions are perceived, and which methods are possible. Method productions, like action productions, cannot fire by themselves, but must be chosen by the control productions.

The **top-level** rules are the only productions which may fire by themselves. As such, they constitute the actual OPS5 production system. They are constructed in such a way that no more than one top-level rule can instantiate in any cycle, thus bypassing the OPS5 conflict-resolution procedure completely.

The following figure illustrates the relationships of the various classes of rules:



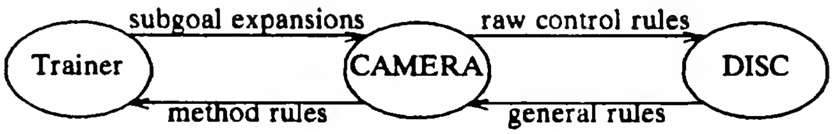
In the above diagram, the set of action rules includes the sensing rules, since the action and sensing rules are conceptually the same. Action rules can be used as sensing rules. For example, the trainer's input "repeatedly put blocks in the box until not put blocks in the box" causes the system to fire the rule "put blocks in the box" until there are no more instantiations of that rule in the conflict set.

The top-level rules, which are not shown, implement the rule hierarchy. These rules constitute a small, fixed set of rules which state simply: fire all the tree rules possible to update the goal tree, then either fire a control rule, or an interface rule if this is a training session. This mechanism could have been programmed into the system, but was implemented as productions in order to minimize the changes to OPS5.

The interface and training rules are considered conceptually to be method rules. Ideally, the process of obtaining help from a trainer should be invoked only when the system decides that is the best method of solving a subgoal. However, in order for this method to be demonstrated by a trainer, it would be necessary for the system to watch a human trainer obtain help from another trainer. For simplicity of implementation, the system was programmed to operate in two separate modes: solve the problem by itself, or watch a human trainer.

3.2.3. Interaction with the Trainer

The overall flow of the system is as follows. At each cycle of a training session, the (human) expert is presented with the set of applicable method productions. If the expert chooses one of the choices, it is executed, the subgoal expansion produces a new goal, and the process repeats. If the expert rejects all the choices, CAMERA constructs, in response to directions from the expert, a new method and a new control production; these productions are added to the system, and DISC is invoked. DISC generalizes the control rule for future use. The following diagram illustrates the interaction of the components:



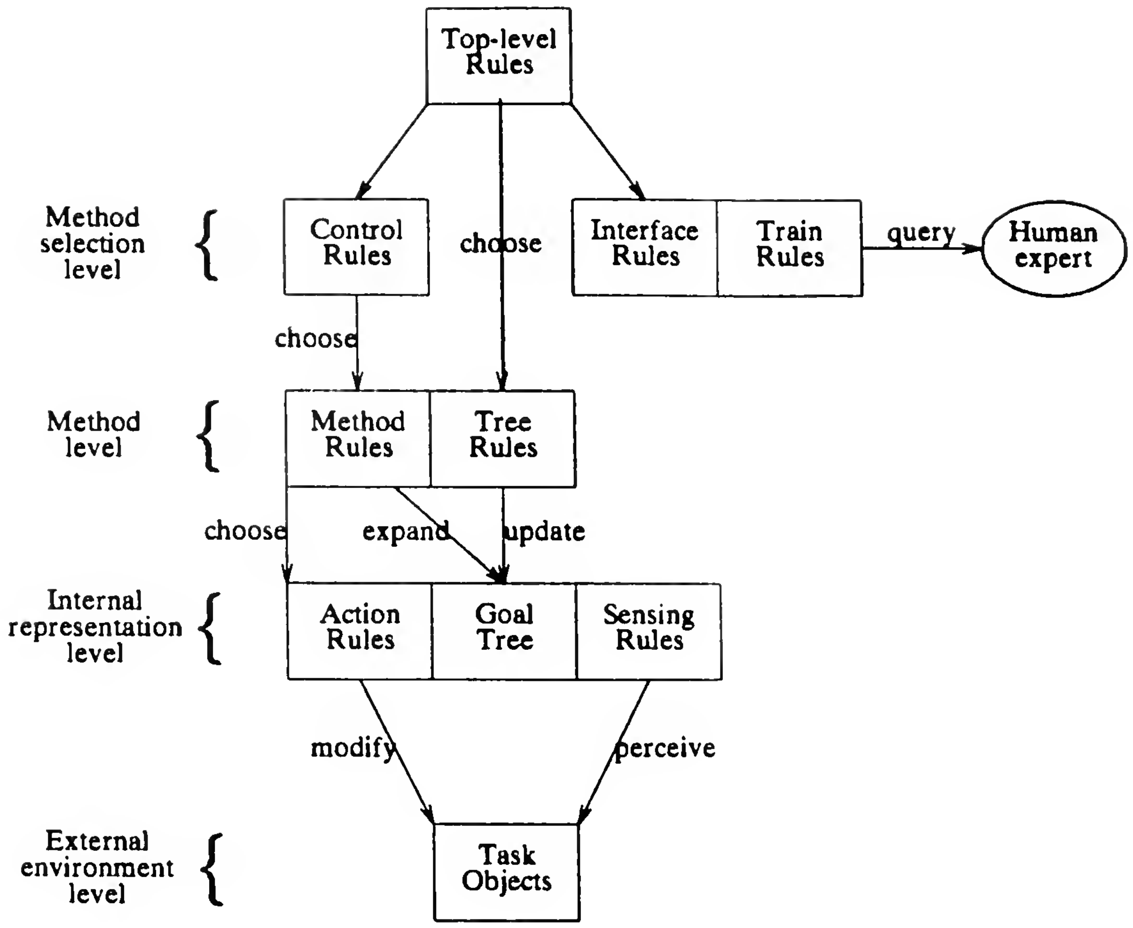
If a trainer wishes to construct a new method production during a training session, CAMERA asks the trainer for the subgoal structure, and asks questions if necessary to determine the parameters and attributes of the subgoals. For example, if the trainer indicates that an action is to be repeated, the interface asks questions to extract the information necessary to build a **repeat** node. This information includes both the type of the action to be repeated (which may itself involve repetition), and the condition for termination of the repetition.

CAMERA then translates the trainer's input into an OPS5 production's right-hand side, and adds the new production to the system's set of method productions. The elements of the left-hand side, (the goal to be expanded and its parameters) are present in working memory, so this new production enters the conflict set. CAMERA causes this production to be fired by OPS5 in the next cycle. In addition, the current conflict set is sent to DISC as the left-hand side of a new control production. The right-hand side is the newly constructed method production.

3.2.4. Overall System Organization

The diagram on the following page illustrates the various components of the system, together with the relationships among the classes of rules. It clearly displays the four levels of knowledge contained in the system:

1) the *control* level, which select the methods;
2) the *method* level, which is composed of both domain-dependent goal expansions and domain-independent goal tree manipulations;
3) the *internal representation* level, which consists of the instantiated action and sensing rules, together with the goal tree;
4) the *environmental* level, which consists of the WM tokens which represent the external task environment.



Conflict set access appears in this diagram in two places: in the control level, as the mechanism by which the methods are selected; and in the internal representation level, in which the goal tokens and the instantiated action-level rules form the state of the task as perceived by the system.

3.3. An Implemented Example

Previously, in Section 2.4, a small production system given in [Sleeman83] which investigates the induction of models of students' skill in algebra, was rewritten using the conflict set representation.

The system described above can be used to implement a goal-directed production system for this task, with the following action rules:

```
(p EVAL action
  (lhs M op N rhs) → (lhs (evaluate M op N) rhs) )

(p SOLVE action
  (SHD M * X = N) → (SHD X = N/M) )

(p TORHS action
  (lhs ± term lhs2 = rhs) → (lhs lhs2 = rhs ∓ term) )

(p COLL action
  (lhs M*X ± N*X rhs) → (lhs < M ± N > * X rhs) )

(p DIST action
  (lhs M * < N*X ± P > rhs) → (lhs M*N*X ± M*P rhs) )

(p BRACE action
  (lhs < M > rhs) → (lhs M rhs) )

(p TOLHS action
  (lhs = rhs ± term rhs2) → (lhs ∓ term = rhs rhs2) )
```

where "op" represents any algebraic operator. Sleeman's original rules implement a variant of the following algorithm:

1) Distribute any multiplications possible;

2) Move all terms without X to the right-hand side;

3) Move all terms with X to the left-hand side;

4) Collect all terms on the left-hand side;

5) Evaluate all operators possible;

6) Solve for X;

7) Evaluate the division.

Note that the artificial REARRANGE rule is no longer needed. This algorithm can be realized by the use of method rules. The following are a few of the method rules (presented for clarity without pointers to other goal nodes) which illustrate the structure from the root node to the action rule TORHS:

```
(p solve-the-equation method
  (root)
  →
  (make goal ^type solve-the-equation) )
```

```
(p move-terms-then-evaluate method
   (goal ^type solve-the-equation)
  →
   (make seq)
   (make goal ^type multiply-out-all-terms)
   (make goal ^type move-all-terms-without-X-to-rhs)
   (make goal ^type move-all-terms-with-X-to-lhs)
   (make goal ^type collect-all-terms-on-lhs)
   (make goal ^type evaluate-everything)
   (make goal ^type solve)
   (make goal ^type evaluate-the-division) )

(p repeatedly-move-a-term-without-X-to-rhs method
   (goal ^type move-all-terms-without-X-to-rhs)
  →
   (make repeat)
   (make goal ^type TORHS)
   (make until ^pname no-terms-without-X-on-lhs) )
```

Some of the control rules which govern selection of methods are:

```
(p CHOOSE-EVAL control
   (production ^pname EVAL ^v1 <op>)
 - (production ^pname PRECEDENCE ^v1 <op2> ^v2 <op>)
  →
   (call fire-production 1) )

(p CHOOSE-SOLVE control
   (production ^pname SOLVE)
  →
   (call fire-production 1) )
```

```
(p CHOOSE-TORHS control
  (production ^pname TORHS ^v2 <term>)
  (production ^pname phrase-is-a-term ^v1 <term>)
  →
  (call fire-production 1) )
```

These control rules reference sensing rules which detect, in the case of CHOOSE-EVAL, the presence of an adjacent operator with a higher precedence, and in the case of CHOOSE-TORHS, whether a phrase is a well-formed term. Other sensing rules detect when a term is in the left-hand side or right-hand side of the equation, when a factor is well-formed, when parentheses match, etc.

This set of rules contrasts in several ways with Sleeman's original system of rules:

1) There are more rules than in the original system. Altogether, the new system contains nearly thirty rules, instead of the original ten.

2) There are fewer action rules, seven instead of ten. These new action rules are independent rules, which state manipulations which are possible on equations. They do not specify an algorithm, as the original rules do, thus increasing the flexibility of the rules. These rules could, for example, be included verbatim in a system for solving higher-order equations. Control rules restrict the firing of these rules to appropriate situations.

3) The algorithm is explicitly represented in the method and control rules, instead of implicitly represented, as in the original rules. This aids in the process of constructing the algorithm.

4) Concepts which people usually apply to first-order equations, e.g., precedence of operators, terms, factors, left-hand side, right-hand side, matching parentheses, etc. are explicitly present in sensing rules, rather than unstated, as in the original rules. This knowledge is represented separately from the actions and methods.

4. Implementation of CAMERA

This chapter contains details of the implementation of various components of CAMERA, and is not essential to an understanding of the concepts developed in the system. Unless the reader wishes to know the details of the implementation of the goal tree, conflict set access, and the various classes of rules, it is advisable to skip to Chapter 5.

4.1. The Goal Tree

The goal tree in WM is assumed to contain nodes of the following form which specify the desirability of achieving a particular goal:

(goal ^type - ^status - ^when - ^ownid - ^father - ^son - ^brother -)

(goal-param ^goalid - ^ce - ^attr - ^value -)

In addition, there are **repeat, seq(uence), and** (nonsequential), **or,** and **not** nodes. These nodes organize the goal nodes into an AND-OR tree which contains two types of AND node (sequential and non-sequential), OR nodes, and repetition nodes, which specify goals which must be repeatedly satisfied until some instantiation, or boolean expression of instantiations, enters the conflict set. The son, brother, and father values are integers which point to the goals which are below, adjacent to, and above a given node in the goal tree.

Goal-param elements modify goals. For example, the goal "find a block":

```
(goal ^type find-a-block ^status active ^when now ^ownid 73 ...)
```

would be modified by:

```
(goal-param ^goalid 73 ^ce block ^attr color ^value red)
(goal-param ^goalid 73 ^c3 block ^attr height ^value 4)
```

to specify a red block with a height of 4.

4.2. Control Rules

A control rule's left-hand side contains conflict set elements which are of the form:

```
(production pname variable binding variable binding ...)
```

where the "pname" is the name of the rule, "variable" represents the variables that actually occur in the rule, and "binding" represents the values that are assigned to the variables. This is the machinery that permits conflict set elements to be accessed by OPS5 productions. The instantiations that occur in a control rule's left-hand side are of three types of rules: action and sensing rules, method rules, and goal tree sensing rules. Goal tree sensing rules are method rules which have empty right-hand sides. These rules reflect the structure of the goal tree. An example of this type of rule is the rule:

```
(p immediate-why goal-sensing
  (goal ^type <t1> ^ownid <id1>)
  (goal ^type <t2> ^father <id1> ^ownid <id2>)
  → )
```

which detects the parent goal of a goal. This reflects the reason that a goal is in the tree. Such instantiations can be chained together to form explanations by rules such as:

```
(p immediate-why-chain goal-sensing
  (production ^pname immediate-why
     ^v1 <t1> ^v2 <id1> ^v3 <t2> ^v4 <id2>)
  (production ^pname immediate-why
     ^v1 <t2> ^v2 <id2> ^v3 <t3> ^v4 <id3>)
 → )

(p why goal-sensing
  (production ^pname immediate-why
     ^v1 <t1> ^v2 <id1> ^v3 <t2> ^v4 <id2>)
  (production ^pname << immediate-why-chain  why >>
     ^v1 <t2> ^v2 <id2> ^v5 <t3> ^v6 <id3>)
 → )
```

Thus, the decision of which method to use can be based not only upon knowledge of the possible methods and actions and perceptual information, but also upon the structure of the goal tree. Note that the rule "why" is recursive and also depends upon a disjunction of two different rules, "immediate-why-chain" and "why". The disjunction is denoted by the double angle brackets "<< ... >>" within the rule "why".

4.3. Implementation of Conflict Set Access

The production system used is OPS5 [Forgy81], extended as described below to permit control productions to make the conflict resolution decisions.

The original OPS5 system does not have the capability of accessing its conflict set as data for the productions. We have extended the system to allow productions to test the conflict set for a particular instantiation of a production. This allows both "reference by name" and "reference by content", as defined by [Davis80b]. The type of content reference implemented is left-hand side reference; the metarules can directly access only the information on the left-hand sides of rules. Furthermore, a production can cause an instantiation of a production to fire. The mechanism to realize this is as follows:

[1] When any production is initially compiled by OPS5, all variables which occur within condition elements have their names changed to "v1", "v2", "v3", etc. Whenever an instantiation is inserted into the conflict set, an element of the following form is added to WM:

(production ^pname -- ^type - ^v1 - ^v2 - ^v3 - ^v4 - ^v5 - ^v6 -
^v7 - ^v8 - ^v9 - ^v10 - ...)

The "v1" field contains the value of the variable v1 in the instantiation, the "v2" field contains the value of v2, etc. This mechanism allows a rule to reference a specific instantiation of a rule. The 'type' field contains the name of the set of rules to which this rule belongs. This must be declared when the rule is compiled. Whenever an instantiation is removed from the conflict set, the corresponding WM element is removed. For efficiency, the conflict set has been changed to an association list, which contains not only the instantiations, but also the corresponding WM entries. For example, if an instantiation of the production

```
(p close-the-eye action
   (eye ^looking-at <object>)
  →
   (modify 1 ^looking-at nothing) )
```

had <object> bound to the value "table", that instantiation would be represented in working memory by the **production** element:

```
(production ^pname close-the-eye ^type action ^v1 table)
```

where the 'v1' field contains the binding of the variable <object>.

[2] An OPS5 user function "fire-production" is provided that causes a particular instantiation in the conflict set to fire. Fire-production takes as input the number of a **production** element on the rule's left-hand side, and causes the corresponding conflict set element to be OPS5's chosen instantiation during the next OPS5 cycle. For example, in the following rule:

```
(p choose-to-get-the-banana control
   (production ^pname get-the-banana)
   (production ^pname climb-down-from-the-box)
  →
   (fire-production 1) )
```

the first production, which is 'get-the-banana', is chosen to be executed next.

[3] The system distinguishes between two kinds of productions: top-level productions and all others. Top-level productions are detected at compilation by the value of the productions' "type" field, which must have the value "top-level". Examples of top-level rules are given in

section 4.5 . The conflict resolution process has been modified to choose only among top-level productions. All other type of rules - action rules, tree rules, method rules, etc. can be fired only when another rule causes them to fire. A control rule can fire only when chosen by a top-level rule, a method rule can fire only when chosen by a control rule, and an action production can fire only when the goal tree contains a goal whose type is identical to the action production's name. The tree productions which enable method rules to fire action rules are:

```
(p fire-production tree
  (production ^pname <p> ^id <id1>)
  (goal ^type <p> ^status active ^when now ^son nil ^ownid <id2>)
 - (goal-param ^goalid <id> ^interpreted nil)
 - (production ^pname instantiation-disagrees-with-goal
      ^v1 <id1> ^v2 <id2>)
 →
  (modify 2 ^status done)
  (call fire-production 1) )

(p instantiation-disagrees-with-goal tree-sensing
  (production ^pname <any> ^type instantiation-disagreeing-with-goal
      ^v1 <id1> ^v2 <id2>)
 → )
```

where the rule "instantiation-disagrees-with-goal" tests to see if the conflict set contains any instantiation from the set of rules called "instantiation-disagreeing-with-goal". Each rule in this set tests one pair of corresponding attributes of the **production** element and the **goal** element. For instance, there is a rule which tests to see if the v1 attribute differs between the **goal** and **production** tokens, another rule tests to see if the v2 attribute differs, and so on. Note that if all rules are defined to be of type "top-level" then the system behaves exactly as the original OPS5.

[4] Normally, when a production is added to OPS5, it is not possible for it to enter the conflict set immediately, as the production-matching network is modified by changes to working memory, not to production memory. However, the system has been altered to allow new method productions to be instantiated immediately, by rehearsing the WM elements in the method rule instantiation. (Rehearsing means referencing the element solely to cause it to be matched against production memory).

[5] OPS5 user functions have been added to the system to perform such tasks as: parsing the trainer's language into an OPS5 production; translating goal parameters into attributes of a goal node; testing **repeat** nodes to see whether or not they are satisfied; choosing a method rule from the conflict set; and displaying the method rules for the trainer.

4.4. Method Rule Construction

CAMERA is responsible for the construction of the system's vocabulary. Whenever the trainer uses a phrase which CAMERA has not previously encountered, CAMERA asks the trainer to construct a sensing production to define the phrase. For example, if the trainer is solving a chess endgame of king and rook versus king, and types the phrase "the king is checkmated", CAMERA will ask for a definition of the phrase. If the trainer defines the phrase to mean "the rook can move to the square the king is on" and "there is no move for the king", then the system will define the phrase "the king is checkmated" to mean the presence of an instantiation of the rule "the rook can move to the square the king is on" and the absence of an instantiation of the rule "there is a move for the king". If CAMERA then does not contain a definition for either of these phrases, it will continue to query the trainer until all phrases have been defined in terms of its previous vocabulary.

CAMERA contains a simple language parser to process the trainer's input. The trainer's input must be a list, i.e., it must be enclosed in parentheses. There must be no parentheses in the list. Certain key words are used by the parser to separate the trainer's input into goals:

a) The keyword 'then' parses into a SEQUENCE node;

b) The keyword 'and' parses into an AND node;

c) The keyword 'or' parses into an OR node;

d) The keyword 'repeat' or 'repeatedly' or 'do' parses into a REPEAT node;

e) The keyword 'not' parses into a NOT node;

f) The phrase **with** ***attribute*** **of** ***object*** = ***value*** parses into the goal parameter:

(goal-param ^ce object ^attr attribute ^value value)

g) Any phrase not containing a keyword parses into a goal type.

This simple mechanism implements the extraction of descriptions from the expert. A more sophisticated language interface, including transformational grammar, would provide flexibility to the trainer in choosing inputs. With the simple parser which was implemented, the trainer needed to be careful to use identical phrases to represent identical conditions and goals.

For example, if the trainer is solving a jigsaw puzzle, and is asked by the system how to put a piece into the puzzle, he may reply:

(join all the edges to the puzzle then let go of the piece in the puzzle)

The parser converts this into:

(seq (join all the edges to the puzzle) (let go of the piece in the puzzle))

then translates this into a method rule in OPS5 form:

```
(p put-a-piece-into-the-puzzle method
   (goal ^type put-a-piece-into-the-puzzle ^status active ^son nil
      ^when now ^ownid <id1>)
  -->
   (make seq ^ownid (goal 1) ^father <id1> ^son (goal 2))
   (make goal ^type join-all-the-edges-to-the-puzzle ^father (goal 1)
      ^ownid (goal 2) ^brother (goal 3))
   (make goal ^type let-go-of-the-piece-in-the-puzzle ^father (goal 1)
      ^ownid (goal 3))
   (modify 1 ^son (goal 1))
   (goalinc) )
```

where (goal n) evaluates to *n* more than the previously highest numbered goal, and (goalinc) updates the variable containing that value.

4.5. Top-level Rules

These rules perform the conflict resolution at each cycle of the system. CAMERA's conflict resolution algorithm is as follows. CAMERA first asks whether it will be trained, or will run independently. Then, CAMERA updates information in the tree. Then, CAMERA either chooses to fire any interface rules, if it is being trained, or else selects a control rule for firing. Only one control rule instantiates in any cycle, as is explained in Chapter 5. These are the top-level rules CAMERA uses to implement this simple hierarchical conflict resolution:

```
(p choose-train top-level
   (production ^pname <p> ^type train)
 - (production ^type tree)
  -->
   (call fire-production 1) )

(p choose-tree top-level
   (production ^pname <p> ^type tree)
  -->
   (call fire-production 1) )

(p choose-interface top-level
   (production ^pname <p> ^type interface)
 - (production ^type << tree train >> )
  -->
   (call fire-production 1) )
```

```
(p choose-control top-level
  (production ^pname <p> ^type control)
 - (production ^type << tree train interface >> )
 →
  (call fire-production 1) )

(p no-top-levelrules-active top-level
  (root ^status active)
 - (production ^type << tree train interface control >> )
 →
  (write (crlf) I do not know what to do. (crlf) )
  (halt) )
```

4.6. Tree Rules

Domain-independent rules that contain information relevant to the structuring of the goal tree are implemented as **tree** productions. This information may be high-level, along the lines of [Wilensky81, Sacerdoti77, Hayes-Roth78], or merely housekeeping information. An example of the former is detecting circular subgoals, which could be represented in OPS5 as:

```
(p circular-subgoals tree
  (goal ^type <p> ^ownid <id1> ^son <id2>)
  (goal ^type <p> ^ownid <id2> ^father <id1>)
 → )
```

This production detects the presence of two different instances of a particular type of goal, which are father and son.

An example of a housekeeping tree production is:

```
(p pass-done-to-an-or tree
  (or ^status active ^ownid <x>)
  (<< goal and or repeat seq >> ^status done ^father <x>)
  -->
  (modify 1 ^status done) )
```

This passes goal satisfaction data back up to an **or** node. CAMERA does not currently implement high-level schemes of goal interaction, costs, and expectations along the lines of [Wilensky81, Sacerdoti77, Hayes-Roth78], but instead concentrates on the issues of representation, ease of extraction, and generalization of control knowledge.

4.7. Interface Rules

If CAMERA is in training mode, it displays the instantiated method rules to the trainer and asks for either a choice, or a new method rule. These are the rules that CAMERA uses to implement the interface control:

```
(p interface1 interface
  (<< goal root >> ^status active ^son nil ^when now)
  (ops-training)
  -->
  (call show-rules) )
```

```
;interface2 chooses an existing method rule
 (p interface2 interface
   (ops-training)
   (ops-max-m-rules ^max { <m> <> 0 } )
   (ops-m-rule ^number { <n> > 0 < <m> } )
  -->
   (remove 2 3)
   (call choose-m-rule <n>) )

;interface3 chooses to build a new method rule
(p interface3 interface
   (ops-training)
   (ops-max-m-rules ^max { <m> <> 0 } )
   (ops-m-rule ^number <m>)
  -->
   (remove 2 3)
   (call build-new-rule) )

;interface4 handles an illegal choice
(p interface4 interface
   (ops-training)
   (ops-max-m-rules ^max { <m> <> 0 } )
   (ops-m-rule ^number { <= 0 } )
  -->
   (remove 3)
   (write (crlf) Type the number of the option you choose: )
   (make ops-m-rule ^number (accept)) )
```

```
;interface5 handles an illegal choice
(p interface5 interface
  (ops-training)
  (ops-max-m-rules ^max { <m> <> 0 } )
  (ops-m-rule ^number { > <m> } )
 →
  (remove 3)
  (write (crlf) Type the number of the option you choose: )
  (make ops-m-rule ^number (accept)) )

;interface6 builds a new method rule since
; no method rule expand the current goal
(p interface6 interface
  (ops-training)
  (ops-max-m-rules ^max 0)
 →
  (remove 2)
  (call build-new-rule) )

(p interface7 interface
  (ops-training)
  (ops-max-m-rules ^max { <any> <> 0 })
 - (ops-m-rule ^number <any2>)
 →
  (write (crlf) Type the number of the option you choose: )
  (make ops-m-rule ^number (accept)) )
```

4.8. Handling of REPEAT nodes

Iteration in the goal tree is handled by **repeat** nodes. A **repeat** node requires the goal immediately below it to be repeatedly satisfied until some condition is met. This is implemented by deleting the subtree under the goal under the **repeat** node, and rebuilding it as long as the condition is not met.

Every **repeat** node contains a pointer to the goal subtree which is to be repeated, and a pointer to a subtree which represents the condition for terminating the repetition. In addition, the **repeat** node contains fields indicating the status of the **repeat** goal ('done','unsatisfied' or 'active'), and the standard pointers to the father and brother goals.

The portion of the goal tree that is under the **repeat** goal is removed after each iteration, and regenerated. This is done to allow each iteration to be accomplished by different means, if necessary. This removal is handled by executive productions.

The deletion of the subtree is done by the following two tree productions:

```
(p delete-node tree
   (delete-this ^id <id1> )
   (<< and or goal repeat seq >> ^ownid <id1>
       ^son <s> ^brother <b> )
  →
   (remove 1)
   (remove 2)
   (make delete-this ^id <s>)
   (make delete-this ^id <b>) )

(p delete-nil tree
   (delete-this ^id nil)
  →
   (remove 1) )
```

The termination condition of a **repeat** node is represented by a subtree composed of **and, or** and **not** nodes. The leaf nodes represent a particular instantiation in the conflict set. The tree represents the logical relation of those instantiations that must be true at the completion of the goal under the **repeat** node. Every node in the logical tree has a 'dormant' status. This prevents the expansion of the goals in the logical tree during the expansion of the rest of the goal tree. At the completion of the expansion and execution of the goal tree under the **repeat** node, the condition represented by the logical tree is tested. If the condition is true, the **repeat** node is marked 'done'. Thus, the **repeat** node corresponds to a **'repeat-until'** construct.

For the purpose of efficiency, the testing of the logical condition is not handled by productions. This would involve many productions, and consume many cycles. Instead, when a **repeat** condition is first tested, the tree is traversed without evaluation, and converted into a corresponding s-expression by the system. This s-expression is then evaluated to determine whether the repetition continues or not. This s-expression is then saved in a list, and is used whenever this **repeat** node is completed again.

The following example illustrates the above features.

```
(p g-warm-up-room method
  (goal ^type warm-up-room ^id <id1> ^son nil
     ^status active)
 →
  (make repeat ^father <id1> ^status active
     ^ownid (goal 1)  ^son (goal 2)
     ^terminate (goal 3) )
  (make goal ^type raise-one-degree ^status active
     ^father (goal 1) ^ownid (goal 2) )
  (make goal ^type temperature-is-comfortable
     ^ownid (goal 3) ^father (goal 1)
     ^status dormant)
  (modify 1 ^son (goal 1)))
```

When the goal of warming up a room is established in the goal tree, the above method production establishes that this can be accomplished by repeatedly raising the temperature one degree until the sensing production 'temperature-is-comfortable' enters the conflict set. This production might appear as:

```
(p temperature-is-comfortable sensing
  (temperature ^value { <v> > 67  < 73 })
 → )
```

Note that this production has a null right-hand side. It serves merely to sense a condition and control the loop.

5. Generalization of Control Information

An important aspect of a knowledge acquisition program such as CAMERA is the ability of the system to extrapolate known data to new situations. This process, called *inductive inference,* permits a system to replace a collection of facts with a single statement. One of the most important ways in which this can be accomplished is by generalization of a group of instances to a class. The importance of generalization is widely recognized, and the problem has been approached by many researchers [Dietterich81, Hayes-Roth78, Kline81, Laird84, Michalski83, Mitchell82, Vere78, Winston75]. The development of a successful approach to generalization would permit the automatic construction of knowledge bases, and thus is the crucial element of any system such as CAMERA. The type of information from CAMERA which must be generalized is the information describing the situations in which alternative methods were selected. This section describes some shortcomings in previous approaches to inductive learning, and proposes remedies.

5.1. Background

Much of the work on generalization is summarized very well in [Dietterich81]. Of particular note are the papers by Mitchell and Vere. In this paper we will use the notation of OPS5 [Forgy81], but the results are applicable to other notations. In this notation, each object is described by a list consisting of a *type* designation, such as "block" or "table", together with a fixed number of *attributes,* such as "color", or "length", each of which has a *value*. Note that in [Forgy81] the term "class" is used instead of "type", which is used here to avoid confusion with other meanings of "class". Attributes are preceded by the symbol "^".

The basic problem of generalization can be described as follows: Given a set of objects:

$$(\text{type}\ \hat{}a_1\ \rho_{11}\ \hat{}a_2\ \rho_{12} \ldots \hat{}a_n\ \rho_{1p})$$
$$(\text{type}\ \hat{}a_1\ \rho_{21}\ \hat{}a_2\ \rho_{22} \ldots \hat{}a_n\ \rho_{2p})$$
$$\ldots$$
$$(\text{type}\ \hat{}a_1\ \rho_{m1}\ \hat{}a_2\ \rho_{m2} \ldots \hat{}a_p\ \rho_{mp})$$

of *positive* instances of the class, i.e., those that are members of the class, and a set of objects:

$$(\text{type}\ \hat{}a_1\ \tau_{11}\ \hat{}a_2\ \tau_{12} \ldots \hat{}a_n\ \tau_{1p})$$
$$(\text{type}\ \hat{}a_1\ \tau_{21}\ \hat{}a_2\ \tau_{22} \ldots \hat{}a_n\ \tau_{2p})$$
$$\ldots$$
$$(\text{type}\ \hat{}a_1\ \tau_{n1}\ \hat{}a_2\ \tau_{n2} \ldots \hat{}a_p\ \tau_{np})$$

of *negative* objects of the class, i.e., those that are not members of the class, derive a description of the class of objects. Typically, a class description is a predicate which depends upon the values of one or more attributes, and evaluates to "true" on all the positive instances, and "false" on all the negative instances.

In the above description of the generalization problem, the set of negative instances may be empty, as it is possible to attempt generalization without the use of negative instances, as in [DeJong83], who used large amounts of domain knowledge to formulate accurate generalizations. However, the difficulty of this approach was recognized by [Winston75], who used negative examples to closely delimit the set of positive examples, thus preventing overgeneralization. The need for negative examples to prevent overgeneralization is also recognized and explained in [Laird84].

Generalization is often cast as the process of transforming class descriptions through the application of generalization rules. Many of the major generalization rules are summarized in [Dietterich81] and [Michalski83]. The rules which appear to be most widely used are:

1) **Variable Substitution**, which replaces a constant value of an attribute with a variable;

2) **Intersection**, which generates a description which includes those attribute-value pairs which are present in all the positive examples and drops those attribute-value pairs which are not present in all the positive examples;

3) **Internal Disjunction**, which expands the set of values which an attribute may assume. This includes "interval filling" for linear attributes, in which all values between two known constant values are assumed, and "domain tree climbing", in which two or more known constant values are replaced by their common ancestor in a hierarchical classification.

4) **Dropping Condition**, which removes a conjunctive condition from a description;

5) **Turning Conjunction into Disjunction**, which replaces the conjunction between two conditions in a description with a disjunction;

6) **Inductive Resolution,** which uses the resolution principle to replace expressions in a class description of the form:

$$(P \Rightarrow F_1) \wedge \neg P \Rightarrow F_2)$$

where P is an attribute-value pair, with the expression

$$(F_1 \mid F_2).$$

Expressions of the form:

$$(A \Rightarrow B)$$

usually occur in the form:

$$(\neg A \mid B)$$

7) **Extension Against,** which replaces an entire description of a class by one discriminating term. This rule classifies instances as belonging to a specific class if one particular attribute does not take on certain values. This is the fundamental generalization rule in the AQ11 system [Michalski78].

8) **Generating Chain Properties,** which characterize instances of a transitive relation which form a chain. An example of a chain is a tower of blocks. Some useful characteristics of a chain of instances are the length of the chain, the first instance, the last instance, the middle instance, etc.

9) **Counting Attributes,** which counts the number of attributes of an instance which satisfy various conditions. For example, the counting attributes rule may count how many numeric attributes of an instance

have values that are between 5 and 8. This rule can generate a large number of new descriptions.

10) **Detecting Descriptor Interdependence,** which detects dependencies among attributes. This has been done by detecting monotonic correspondences, and by calculating the correlation coefficient between two attributes [Michalski83]. However, other types of relationships among attributes have not been detected.

Note that all of these rules except the last two test single attributes, not conjunctions of attributes. This is an important observation, as we will show in Section 5.2.1.1 that it imposes severe limitations upon the capability of programs which use these rules.

5.2. Generalization by Discrimination and Description

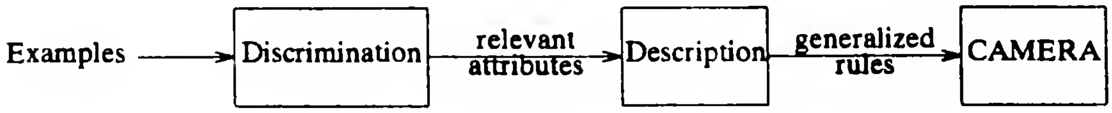
5.2.1. Discriminatory Learning

It has been postulated [Michalski83] that there are two different types of descriptions which are useful in inductive learning: *characteristic* descriptions, which contain all characteristics common to all positive instances; and *discriminatory* descriptions, which contain those characteristics which are absent from all negative instances. Characteristic descriptions have been frequently used in research on inductive learning [Hayes-Roth78, Laird84, Mitchell77, Vere78], and discriminatory descriptions, which have a long history [Fisher36, Nilsson65], have recently achieved widespread use [Brazdil78, Langley83, Michalski78, Michalski80, Vere80].

In order for a system such as CAMERA to be useful, it must be able to generalize the raw control rules which are constructed during training sessions. Otherwise, acquired knowledge can be applied only to situations which are identical to those previously encountered. This is discriminatory learning, since it requires the construction of a description of the conditions under which each method rule is chosen from the set of all possible method rules for expanding a particular goal. The task of control rule generalization is simpler than the general generalization problem. This simplification is due to the fact that method rules occur in sets, and control rules choose one member of that set. When a goal of a particular type is activated, all the method rules which expand goals of that type (with the given types of parameters) instantiate. Either all the method rules in a set instantiate, or none do. Thus, the task of control rule generalization becomes the task of classifying conditions for method selection.

Some previous discrimination algorithms [Brazdil78, Langley83] have constructed their descriptions by searching for a condition which is present only in the positive examples and not in the negative examples, or for a condition which is present only in the negative examples and not in the positive examples. This seems an intuitive strategy, but it has three major shortcomings, which are described in the following sections.

To attempt to remedy these shortcomings, DISC models generalization as a process consisting of two separate steps: discrimination, and description.



5.2.1.1. Functional Dependencies Among Attributes

One of the shortcomings in the discrimination algorithms used in [Brazdil78, Langley83] is that they depend upon the detection of single conditions which separate the positive examples from the negative examples. However, in complex situations, this may not be possible. It may be that a conjunction of conditions must be found in order to separate the two sets of examples, i.e., that a function of those attributes determines whether an instance is positive or negative. This motivates the following definition.

Definition:

We define a set of attributes $\mathcal{A} = \{a_1, \dots a_n\}$ to be *discriminatory* if there is a function $f(a_1, \dots a_n)$ which evaluates to 1 on positive instances and evaluates to 0 on negative instances.

One way to attempt to handle such sets of attributes is to presume the existence of predicates which detect particular conjunctions of attributes. [Michalski83] terms these predicates "constructive generalizations", as they represent new, complex attributes which are constructed from simple attributes. However, as [Michalski83] notes, existing systems do not actually construct these predicates, but assume a set of them as part of the given generalization language. Thus, if these systems do not already possess the necessary predicates, they will be unable to efficiently construct an accurate description.

To address this problem, it is first necessary to develop a framework within which algorithmic and heuristic approaches to discrimination can be discussed. In this work, the discriminatory functions that are constructed consist only of conjunctions of attributes. This choice was made because it permits simplicity of implementation without sacrificing logical completeness of formulation. To express disjunctions of conditions, DISC uses conjunctions, together with negation. For example, if the desired description of the set of positive instances is:

$$A \mid B$$

then DISC will construct the description:

$$-A \ \& \ -B$$

for the set of negative instances.

The use of conjunctive descriptions rather that descriptions which employ both conjunctions and disjunctions leads to a greatly reduced search space for the trainer. The problem of detecting conjunctions of attributes can be represented as search within the power set of the set of attributes. If there are n attributes, then there are 2^n conjunctions of attributes which may separate the positive and negative examples. This is much smaller than the 2^{2^n} disjunctions of conjunctions of attributes. Since the trainer inputs only a list of the attributes which he feels are discriminatory, there are 2^n such lists from which to choose, given n possible attributes. The system must then examine all the instances to see whether the input conjunction of attributes discriminates among the positive instances and negative instances. This requires ml comparisons, where m is the number of instances, and l is the number of attributes input by the trainer.

Exhaustive search of the space of conjunctions is possible only for problems whose element descriptions are short. If, for example, a description consists of only ten attributes, then the resulting power set is manageable. Constructive generalizations serve to shorten descriptions, thus enabling exhaustive search in some cases in which it would otherwise be too expensive. However, in general, exhaustive search proves too expensive. Thus, it would be necessary to search this space heuristically. The construction of heuristics which propose conjunctions of attributes would be the heart of the generalization process within this paradigm.

However, for a system to be able to learn, it must first be able to be told. As this system has access to the expertise of a human trainer, and as this work focuses on extraction of knowledge from human trainers, the system does not automatically construct these heuristics, but instead extracts conjunctions of attributes from the advice of the trainer in the following manner:

1) If the raw control rule that CAMERA produces from the trainer's input expands a new type of goal, DISC does not generalize the raw control rule, but preserves it as a positive instance of the selection of the method used to expand the goal. In this initial case, since there are no negative instances, DISC has no discriminatory criterion, and hence no restrictions on when to apply this method, i.e., since the system has never seen this type of goal before, and contains only one method for expanding it, this method must always be chosen. DISC terminates in this case.
2) If the raw control rule expands a previously known type of goal, and agrees with the existing discriminatory criterion (conjunction of attributes), then DISC leaves the criterion unchanged, and terminates.
3) If the raw control rule disagrees with the existing discriminatory criterion, then DISC presents the existing criterion to the trainer, together with the goal and method, and asks the trainer to modify the criterion by either removing an existing conjunction of attributes, or inserting a new conjunction of attributes. After the trainer has modified the criterion, if the new criterion successfully separates the positive and negative examples, then DISC terminates. Otherwise, DISC informs the

trainer of the failure, and displays one of the instances which were not grouped properly (either a positive instance which would be excluded by the modified criterion, or a negative instance which would be included.) This process is repeated until the trainer produces a criterion which separates the sets of instances. Note that modifying a selection criterion in this way may require the modification of existing selection criteria for other methods for the same goal, by replacing existing criteria with the set difference of the old and new criteria. This is necessary to maintain the disjoint nature of the control rules.

The attribute conjunctions which are extracted from the trainer are retained by the system as constructive generalizations, implemented as sensing rules. In this way, the descriptive power of the system is enhanced. Note that this expansion of the system's rule set can cause inconsistencies between the new instances and the existing instances. For example, if a new sensing rule is built, which describes a conjunctions of conditions which occur in all the positive instances and in none of the negative instances, this sensing rule should instantiate in all future positive instances. But this rule did not exist when past positive instances were encountered, and thus does not appear in any past positive instances. This inconsistency can cause DISC to reject the new sensing rule as discriminatory, since it does not appear in all the positive instances. To remedy this, DISC creates an updated version of all instances while it is examining them in step 3 above to determine whether the trainer's criterion discriminates between the positive and negative instances. If the criterion turns out to be discriminatory, the updated version of the instances replaces the existing instances.

As is illustrated in Chapter 6, the above interaction between the system and the trainer requires less detailed input from the trainer than, for example, a system such as TEIRESIAS [Davis77], which requires the trainer to actually debug the rule set.

5.2.1.2. Models of Discrimination

A second shortcoming in the type of discrimination described in [Michalski78] is that simply detecting conditions, or conjunctions of conditions, which are true in one set of instances and never true in the other set of instances, requires the presentation of a very carefully prepared set of training instances, in order to closely delimit the set of positive instances. These "near misses" [Winston75] serve to eliminate many attributes as discriminators, thus clearly displaying those attributes which separate the sets of instances. The necessity of near misses is a severe shortcoming, as there is, in general, no guarantee that data will contain sufficient near misses. Furthermore, if the discriminating conjunctions of attributes contains n attributes, then a near miss must be available for every subset of the conjunction which contains $n-1$ elements. This is a difficult requirement to meet.

For example, suppose a program is presented with the following instances of cars that were bought by a dealer:

```
(car ^color green ^mpg 26 ^avgrep 200 ^price 8500 ^length 165 ^height ...)
(car ^color red   ^mpg 18 ^avgrep 100 ^price 9800 ^length 173 ^height ...)
(car ^color blue  ^mpg 30 ^avgrep 300 ^price 8300 ^length 178 ^height ...)
```

together with the following instances of cars which were not bought:

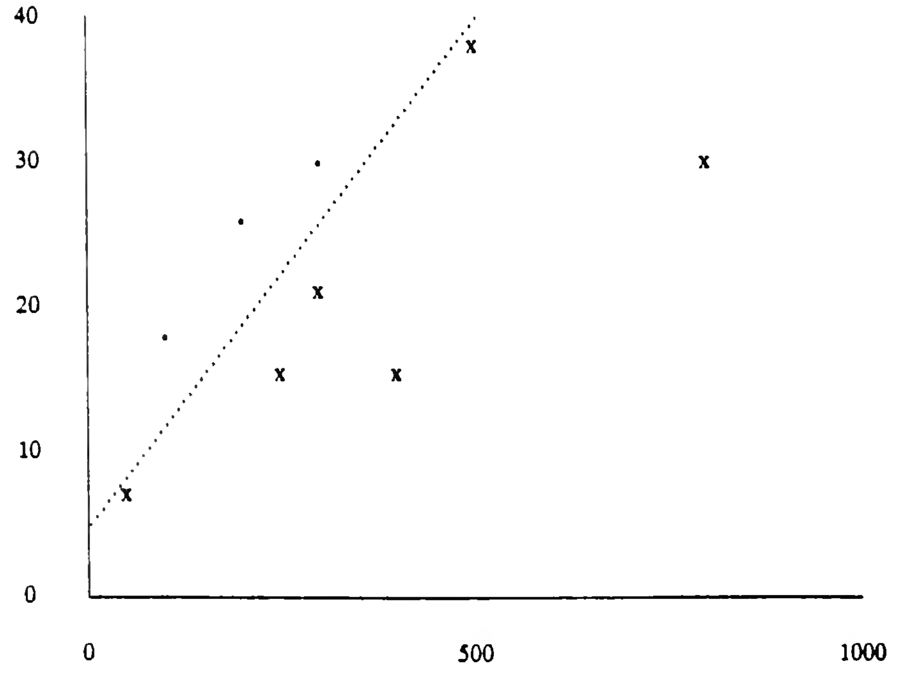
```
(car ^color red   ^mpg 35 ^avgrep 800 ^price 7025 ^length 180 ^height ...)
(car ^color blue  ^mpg 14 ^avgrep 400 ^price 8475 ^length 175 ^height ...)
(car ^color white ^mpg 14 ^avgrep 250 ^price 9250 ^length 171 ^height ...)
(car ^color blue  ^mpg  7 ^avgrep  50 ^price 7550 ^length 170 ^height ...)
(car ^color red   ^mpg 28 ^avgrep 500 ^price 9640 ^length 166 ^height ...)
(car ^color black ^mpg 21 ^avgrep 310 ^price 9335 ^length 182 ^height ...)
```

and must infer the reasons for the dealer's purchases. A discrimination algorithm which simply detects those conditions which are true in one set of instances and not in the other could possibly arrive at an incorrect description of the reason that the cars were purchased. Such an algorithm could conclude that only cars with prices that are a multiple of one hundred should be purchased, or that cars whose gas mileage (mpg) is not a multiple of seven should be purchased, or that cars whose average repair cost in a year (avgrep) is exactly 100, 200, or 300 should be purchased. If near misses were present, then such an error could be avoided. But since we cannot dictate the parameters of automobiles to the manufacturers, near misses do not necessarily exist.

The underlying cause of such a problem is that the model of discrimination is too simple. A model which can recognize patterns in the data is needed. Michalski's STAR algorithm [Michalski83] deals with this problem very effectively, but requires that a descriptive language be provided which contains a sufficiently rich set of descriptors to describe similarities and differences between instances. For example, in the task of chess, this language must be able to describe a position not only in terms of what pieces are on which squares, but also in terms of the rows, columns, diagonals, pawn structures, king's safety, etc. Such a language is difficult and time-consuming to construct anew for each task. CAMERA uses conflict set descriptions as the language to describe instances, and DISC looks for patterns in these descriptions.

For DISC, one of the models of pattern recognition which was chosen was that of *linear separation* [Nilsson65, Hunt75]. Linear separation attempts to insert a hyperplane between the positive instances and the negative instances, and is therefore applicable only to *ordered* attributes, i.e., attributes whose values have an intrinsic ordering. When a conjunction of n ordered attributes is proposed as a discriminator, then DISC attempts to insert an $n-1$ dimensional hyperplane between the positive and negative instances. If the sets of positive and negative instances can be separated in this way, then the conjunction of attributes is accepted as a discriminator. Otherwise, it is not accepted. For attributes which are not ordered, DISC uses the model of absolute discrimination, i.e., those attributes' values can occur only in one set of instances and not in the other set of instances.

In the above example, the pattern recognizer rejects all the individual attributes as discriminators, even though identical values are present in both sets only for the color attribute, since neither the positive instances' values nor the negative instances' values form a connected cluster. Thus the examination of conjunctions of two attributes is necessary. When this space is searched, the conjunction that satisfies the clustering criterion is that of mileage (mpg) and average yearly repair cost (avgrep). The relationship between these two attributes can be displayed as:



with the dotted line representing one possible separation boundary between the positive and negative instances.

Note that the pattern recognizer serves as a pattern "checker" rather than a pattern "generator", as it is presented with points which have already been classified. This allows the possible use of multiple pattern recognition models which would function as demons, in parallel if possible. If any of these models detected an understandable pattern, then the conjunction of attributes would be accepted as discriminatory. For simplicity, DISC uses the single model of linear separability.

5.2.2. The Description of Generalizations

After finding a conjunction of attributes which discriminates between the positive and negative instances, the function which relates these attributes must be formulated, i.e., we must decide upon a function which contains the data points. Thus, it becomes necessary to discuss the method of representing generalizations, i.e., it is necessary to describe the ***generalization language***.

As the situations which are to be generalized are described in terms of instantiations from the conflict set, the generalization language must be able to refer to instantiations and their variable bindings. Also, the description of the discriminating attributes is necessarily constrained by the pattern output by the pattern recognizer. As the pattern recognizer implemented within DISC detects linear separability, the generalization language needs to be able to represent linear combinations of values, and to express inequalities with them.

The current version of the generalization language is very similar to VL_1 [Michalski83], with references to variable bindings of instantiations allowed. Only conjunctions of descriptors are permitted. The four basic arithmetic operators can be used, as can the relational operators: $<, >, =, \neq, \geq, \leq$. Generalization descriptions are compiled into production form, so that they can be referenced by other rules.

The following are examples that show how the system represents discriminatory conditions:

1) In the previous example of purchasing cars, DISC would represent the criterion for purchasing a car as:

$$35 * avgrep / 500 + 5 - mpg \leq 0$$

where avgrep and mpg are the values of those attributes. This is the equation of the separation boundary between the positive and negative instances. Due to the inability of OPS5 to compute expressions on the left-hand side of a rule, DISC implements this criterion by the three rules:

```
(p criterion-to-purchase-car control
  (production ^pname purchase-car ^v1 <i>)
  (production ^pname ignore-car ^v1 <i>)
  (car ^mpg <m> ^avgrep <a> ^id <i>)
  →
  (make criterion ^id <i> ^value
      (compute ( 35 * <a> ) / 500 + 5 - <m> )) )

(p choose-to-purchase-car control
  (production ^pname purchase-car ^v1 <i>)
  (production ^pname ignore-car ^v1 <i>)
  (criterion ^id <i> ^value { <v> <= 0 } )
  →
  (call fire-production 1) )
```

```
(p choose-to-ignore-car control
  (production ^pname purchase-car ^v1 <i>)
  (production ^pname ignore-car ^v1 <i>)
  (criterion ^id <i> ^value { <v> > 0 } )
  →
  (call fire-production 2) )
```

2) In the example of deciding how to put a piece into a jigsaw puzzle, CAMERA might contain an action rule such as:

```
(p pick-up-the-piece action
  (hand ^contents <n>)
  (eye ^looking-at <id1>)
  (piece ^id <id1> ^color <c1> ^location <loc1>)
  →
  (modify 1 ^contents <id1>) )
```

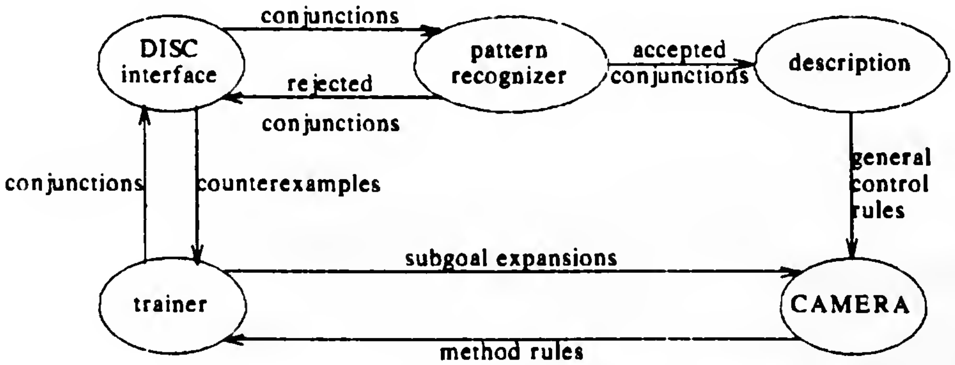
In this case, DISC could represent one possible condition for deciding which piece to pick up as follows:

```
(p choose-a-piece-with-the-same-color control
   (production ^pname pick-up-the-piece ^v2 <id1>
              ^v4 <c1> ^v5 in-the-heap)
   (production ^pname pick-up-the-piece ^v2 <id2>
              ^v4 <c1> ^v5 in-the-puzzle)
  -->
   (call fire-production 1) )
```

which reflects the decision to pick up a piece in the heap of pieces which has the same color as a piece already in the puzzle.

5.2.3. Summary of the System Structure

DISC models learning as a process consisting of two distinct steps: discrimination, then description. In the discrimination step, DISC queries the trainer for a conjunction of discriminating attributes, as described above, and checks it by processing the conjunction of attributes with a clustering algorithm, to see whether the trainer's proposed conjunction actually separates the positive and negative instances. The implemented checking pattern recognizers are linear separation, and binary separation. Once DISC has a satisfactory discriminatory criterion, it then proceeds to describe it in terms of the conflict set description language. The resulting generalized control rule is used by CAMERA to select methods when the system is running independent of the trainer. The following diagram displays the flow of control in the system:



5.2.4. Implementation of DISC

The ability of the modified OPS5 production system to access the conflict set, together with the use of production types, allows the easy implementation of hierarchical production sets, in which the conflict resolution of a set of productions can be controlled by productions in a set which is higher in the hierarchy. DISC is implemented as such a set of productions within CAMERA. DISC is activated whenever a raw control rule has been constructed during a training run of CAMERA, i.e., immediately after the **interface** productions have fired.

5.2.4.1. Implementation of Discrimination

Due to the inability of OPS5 to manipulate the forms of its own rules, DISC must represent CAMERA's rules as working memory tokens. DISC is implemented as a separate OPS5 application. WM contains:

[1] Tokens representing the method rules. These represent the structure of the rules. Each token represents one node in a method rule; there is one token for the goal which is expanded, which is on the left-hand side, and a token for each node in the subgoal expansion, which is on the right-hand side of the method rule. These tokens are linked exactly as are the goal nodes in the right-hand side of the method rule.

```
(method-rule ^name ... ^side left/right ^node-type ... ^father ...
        ^son ...)
```

[2] Tokens representing the names of the control rules.

```
(control-rule ^name ...)
```

[3] Pointer tokens which connect control rule names with method rule tokens. These may be "positive", signifying that the method rule is present on the left-hand side of the control rule, or "negative", signifying

that the method rule is present in negated form. These pointer tokens form the structure of the control rules, by organizing the method rules to form the control rule left-hand sides.

(pointer ^control <cr> ^method <m> ^present positive/negative)

[4] Raw control rules, each of which consists of a situation from CAMERA, represented by the conflict set, together with the method rule the trainer chose. The conflict set elements are represented by **production** tokens, and are linked together by **pointer** tokens, exactly as in [3].

The generalization is performed by a set of productions which implement an interface which queries the trainer for a conjunction of attributes, test if it discriminates the positive instances from the negative instances, and then insert **pointer** tokens to connect discriminatory attributes to a control rule name token to create a new control rule.

The discrimination algorithm makes use of sensing productions, which detect conditions such as:

- Whether the type of the expanded goal is known or new;

- Whether an attribute has a value which occurs in both a positive and a negative instance;

- Whether a conjunction of attributes has a tuple of values which occurs in both a positive and a negative instance;

- Whether a goal's discriminatory criterion does discriminate between the positive instances and the negative instances, after the new instance has been added.

5.2.4.2. Implementation of the Pattern Recognizer

As previously described, the type of pattern recognition that DISC uses is linear separability [Nilsson65, Hunt75]. The algorithm searches for a hyperplane which separates the positive and negative instances, and consists of the following steps:

1) Choose an initial hyperplane;
2) Repeatedly modify this hyperplane according to those data points which are misclassified, until all points are correctly classified by the hyperplane;
3) If the average number of points misclassified does not decrease over a number of iterations equal to the number of data points, then stop iterating. In this case, the algorithm judges that the points are not linearly separable, since the number of misclassifications does not decrease after one pass through the entire data set. This termination bound on the iteration process was chosen to allow larger problems more time to find a solution, while preventing the algorithm from taking too much time.

The representation of the n dimensional hyperplane is as an $n+1$ tuple, which contains the coefficients of the hyperplane in Euclidean space. As it turns out [Hunt75], the choice of initial hyperplane is not important. The modification process alters the coefficients of the hyperplane, by adding a multiple of the coordinates of misclassified points to the coefficients. The multiple is proportional to the amount of misclassification of the data point. Details of the algorithm may be found in [Nilsson65] and [Hunt75].

This algorithm is implemented as an OPS5 user function, which is called by the DISC interface after the trainer has proposed a conjunction of attributes as discriminatory. The function processes the values of these attributes with the iterative procedure outlined above, and returns either "yes" or "no" into working memory, depending on whether that conjunction of attributes separates the positive and negative instances. If the answer is "yes", then the description process takes the conjunction of attributes, and constructs a generalized control rule for CAMERA. If the answer is "no", then the function also returns a data point which it could not classify correctly, which DISC displays to the trainer, asking for a new conjunction of attributes.

6. Example Tasks

Examples were constructed to get some feel for the kind of method and control productions which would be generated in practice. This chapter contains descriptions of these examples, including traces of the system's interactions with the trainer, and samples of the method rules, raw control rules, and general control rules produced by the system. Further details are given in the appendices at the end of the paper.

6.1. Solving a Jigsaw Puzzle

The first example used was a nondeterministic production system which solved a jigsaw puzzle similar to that in [Stolfo79]. When left to run with no guidance from method or control rules, CAMERA repeats the same sequence of productions, and accomplishes nothing. However, when method productions of the form described above were used to implement a simple strategy, the system proceeded directly to the solution.

The interaction between the trainer and CAMERA is given on the following pages. CAMERA's outputs are given in italics and the trainer's inputs are preceded by ">". As each step is completed, CAMERA informs the trainer that the step is done. Annotations are enclosed in boxes. Note that when CAMERA queries the trainer for discriminatory attributes, it is necessary for the trainer to refer to instantiations, i.e., the trainer's language consists of references to what could or could not be done. Details of the rules and working memory are given in Appendix 1.

What problem would you like me to solve?
Type in the steps to solve the goal:

>(solve a jigsaw puzzle)

Please show me how to solve a jigsaw puzzle
Type in the steps to solve the goal:

>(repeatedly put a piece into the puzzle until the puzzle is finished)

Please show me how to put a piece into the puzzle
Type in the steps to solve the goal:

>(get a piece from the heap then put a piece into the puzzle)

Note the ambiguous definition of the goal type *put-a-piece-into-the-puzzle*. This is a deliberate attempt to confuse CAMERA.

Please show me how to get a piece from the heap
Type in the steps to solve the goal:

>(look at a piece with location of piece = in heap then pick
up the piece in view)

Note that until this point all inputs have produced subgoal expansions. However, this last input corresponds to the action rule *look-at-a-piece* and therefore an instantiation of this rule which has the "location" attribute of the piece equal to "in-heap" is chosen to fire.

get a piece from the heap is done

Choose a course of action to put a piece into the puzzle:

1> get a piece from the heap then put a piece into the puzzle
2> Do something else.

*Type the number of the option you choose:*2

This is the first time that there has been an existing method to expand a goal, so a menu is presented to the trainer.

Please show me how to put a piece into the puzzle
Type in the steps to solve the goal:

>(put the piece down in the puzzle)

What attributes of the situation caused you to choose this method and not another? Please list them:

> pick up the piece in view

The trainer refers to the fact that he cannot pick up the piece in view, as he could for the other method. *Put-the-piece-down-in-the-puzzle* is an action rule, so it is fired.

put a piece into the puzzle is done

Choose a course of action to put a piece into the puzzle:

1> get a piece from the heap then put a piece into the puzzle
2> just put the piece down in the puzzle
3> Do something else.

Type the number of the option you choose:1

The more extensive menu reflects CAMERA's growing set of methods for putting a piece into the puzzle.

What attributes of the situation caused you to choose this method and not another? Please list them:

> pick up the piece in view

Choose a course of action to get a piece from the heap:

1> look at a piece with location of piece = in heap then pick up the piece in view
2> Do something else.

Type the number of the option you choose:2

Please show me how to get a piece from the heap
Type in the steps to solve the goal:

>(find a piece that matches an edge in the puzzle then pick up the piece in view)

What attributes of the situation caused you to choose this method and not another? Please list them:

> look at a piece (piece location in-puzzle)

Please show me how to find a piece that matches an edge in the puzzle
Type in the steps to solve the goal:

>(repeatedly look at a piece with location of piece = in heap
until looking at a matching piece)

find a piece that matches an edge in the puzzle is done

get a piece from the heap is done

Choose a course of action to put a piece into the puzzle:

1> get a piece from the heap then put a piece into the puzzle
2> just put the piece down in the puzzle
3> Do something else.

Type the number of the option you choose:3

Please show me how to put a piece into the puzzle
Type in the steps to solve the goal:

>(join all the edges to the puzzle then let go of the
piece in the puzzle)

What attributes of the situation caused you to choose this method and not another? Please list them:

> ((pick up the piece in view)
  (look at a piece (piece location in-puzzle)))

The trainer refers to two actions which were important in deciding which method to use. Note that the actual boolean condition which DISC will construct will be:

pick-up-the-piece-in-view $\wedge$ look-at-a-piece (...)

Please show me how to join all the edges to the puzzle
Type in the steps to solve the goal:

>(repeatedly join an edge of the piece to an edge in
the puzzle until not edge of the piece matches an edge in the puzzle)

join all the edges to the puzzle is done

put a piece into the puzzle is done

put a piece into the puzzle is done

Completion markers are passed up the goal tree as goals are accomplished. From this point on, no new methods are entered by the trainer. Existing methods are chosen from menus.

Choose a course of action to put a piece into the puzzle:

1> get a piece from the heap then put a piece into the puzzle
2> just put the piece down in the puzzle
3> join all the edges to the puzzle then let go of the piece in the puzzle
4> Do something else.

Type the number of the option you choose:1

Choose a course of action to get a piece from the heap:

1> look at a piece with location of piece = in heap then
pick up the piece in view
2> find a piece that matches an edge in the puzzle then
pick up the piece in view
3> Do something else.

Type the number of the option you choose:2

```
Choose a course of action to find a piece that matches an edge in the puzzle:

1> repeatedly look at a piece with location of piece = in heap
   until looking at a matching piece
2> Do something else.

Type the number of the option you choose:1


find a piece that matches an edge in the puzzle is done


get a piece from the heap is done

Choose a course of action to put a piece into the puzzle:

1> get a piece from the heap then put a piece into the puzzle
2> just put the piece down in the puzzle
3> join all the edges to the puzzle then let go of the piece in the puzzle
4> Do something else.

Type the number of the option you choose:3

Choose a course of action to join all the edges to the puzzle:

1> repeatedly join an edge of the piece to an edge in the puzzle
   until not edge of the piece matches an edge in the puzzle
2> Do something else.

Type the number of the option you choose:1

join an edge of the piece to an edge in the puzzle is done
join all the edges to the puzzle is done

put a piece into the puzzle is done

put a piece into the puzzle is done
```

```
Choose a course of action to put a piece into the puzzle:

1> get a piece from the heap then put a piece into the puzzle
2> just put the piece down in the puzzle
3> join all the edges to the puzzle then let go of the piece in the puzzle
4> Do something else.

Type the number of the option you choose:1

Choose a course of action to get a piece from the heap:

1> look at a piece with location of piece = in heap then pick up
   the piece in view
2> find a piece that matches an edge in the puzzle then pick up
   the piece in view
3> Do something else.

Type the number of the option you choose:2

Choose a course of action to find a piece that matches an edge in the puzzle:

1> repeatedly look at a piece with location of piece = in heap
   until looking at a matching piece
2> Do something else.


find a piece that matches an edge in the puzzle is done

get a piece from the heap is done

Type the number of the option you choose:1
```

```
Choose a course of action to put a piece into the puzzle:

1> get a piece from the heap then put a piece into the puzzle
2> just put the piece down in the puzzle
3> join all the edges to the puzzle then let go of the piece in the puzzle
4> Do something else.

Type the number of the option you choose:3

Choose a course of action to join all the edges to the puzzle:

1> repeatedly join an edge of the piece to an edge in the
   puzzle until not edge of the piece matches an edge in the puzzle
2> Do something else.

Type the number of the option you choose:1


join all the edges to the puzzle is done

put a piece into the puzzle is done

put a piece into the puzzle is done

solve a jigsaw puzzle is done
```

In this example, three method rules existed for the goal of type "put a piece into the puzzle". CAMERA generated raw control rules of the form:

```
( (production ^pname get-a-piece-then-put-it-in ^type method)
  (production ^pname put-the-piece-down ^type method)
  (production ^pname join-all-edges-then-let-go ^type method)
  (production ^pname pick-up-the-piece-in-view ^type action)
  (production ^pname immediate-why ^type goal-sensing)
                        .
                        .  other action and sensing rules
                        .
 →
  (call fire-production 1) )

( (production ^pname get-a-piece-then-put-it-in ^type method)
  (production ^pname put-the-piece-down ^type method)
  (production ^pname join-all-edges-then-let-go ^type method)
  (production ^pname put-the-piece-down-in-the-puzzle ^type action)
  (production ^pname immediate-why ^type goal-sensing)
                        .
                        .
                        .
     other action and sensing rules, not including:
  (production ^pname look-at-a-the-piece ^type action ^v3 in-puzzle)
                        .
                        .
                        .
 →
  (call fire-production 2) )
```

```
( (production ^pname get-a-piece-then-put-it-in ^type method)
  (production ^pname put-the-piece-down ^type method)
  (production ^pname join-all-edges-then-let-go ^type method)
  (production ^pname look-at-a-the-piece ^type action ^v3 in-puzzle)
  (production ^pname put-the-piece-down-in-the-puzzle ^type action)
  (production ^pname immediate-why ^type goal-sensing)
                        .
                        . other action and sensing rules
                        .
  -->
  (call fire-production 3) )
```

For the above raw control rules, DISC produced the following general control rules:

```
(p choose-to-get-a-piece-then-put-it-in control
   (production ^pname get-a-piece-then-put-it-down ^type method)
   (production ^pname put-the-piece-down ^type method)
   (production ^pname join-all-edges-then-let-go ^type method)
   (production ^pname pick-up-the-piece-in-view ^type action)
   -->
   (call fire-production 1) )

(p choose-to-just-put-the-piece-down control
   (production ^pname get-a-piece-then-put-it-in ^type method)
   (production ^pname put-the-piece-down ^type method)
   (production ^pname join-all-edges-then-let-go ^type method)
   (production ^pname put-the-piece-down-in-the-puzzle ^type action)
  -(production ^pname look-at-a-the-piece ^type action ^v3 in-puzzle)
   -->
   (call fire-production 2) )
```

6.2. A Chess Endgame: King and Rook vs. King

To more fully test the capabilities of the system, a more complicated task than the jigsaw puzzle was chosen. The task of king and rook versus king is one which most existing chess programs cannot solve without subroutines specifically programmed for this purpose. As in the puzzle task, the trainer began with an initial state consisting of the chess position and the root node. The first subgoal expansion posed the task of checkmate by attaching a goal of type "checkmate the opponent" to the root. The trainer then proceeded to define this task in more precise terms.

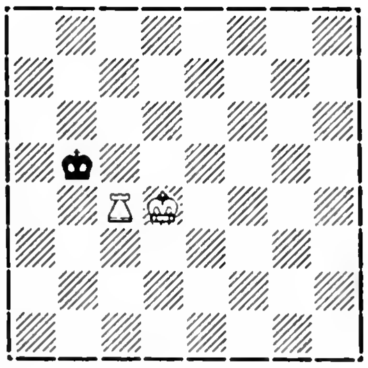
In solving this problem, the trainer expanded the goal "force the king to the edge of the board" into a repetition of the subgoal "reduce the king's space" until an instantiation of the sensing rule "the opponent's king cannot move from the edge of the board" occurs. The goal "reduce the king's space" was expanded in several different ways, depending on the position. For instance, if the trainer's king could be moved toward the opponent's king, then this method was chosen. This choice resulted in two rules: the method rule which expands the goal "reduce the king's space" into "move the king towards the king", and the raw control rule which states that if an instantiation of "king can move to square closer to king" exists, then expand the goal in that fashion. The trainer solved the goal "move the king towards the king" by finding a square closer to the opponent's king to which the trainer's king could move, then picking up the king, moving the hand to the chosen square, and releasing the king. As in the puzzle example, the goal "find a square which is closer to the opponent's king and to which I can move" was solved by repeatedly looking at squares until sensing rules representing the desired properties instantiated.

Another way in which the goal "reduce the king's space" was expanded was: "find the empty rank or file between the kings" then "move the rook to that rank or file". This was executed only if the rook could not be captured on that square.

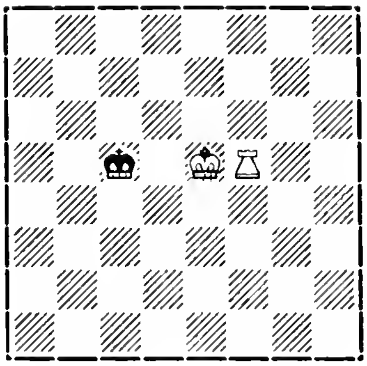
Using these rules, the trainer guided CAMERA until it had pinned the opponent's king against the edge of the board. Then the goal "checkmate the king" was solved by reducing the opponent's king's space to two squares, then moving the king across from the opponent's king, so as to pin it against the edge of the board, then administering mate. Administering mate might involve wasting a move to force the opponent's king into the corner. Reducing the king's space to two squares was accomplished in the same way as "force the king to the edge of the board" was earlier.

In this way, CAMERA was trained to checkmate on four different initial positions of king and rook versus king. These positions were:

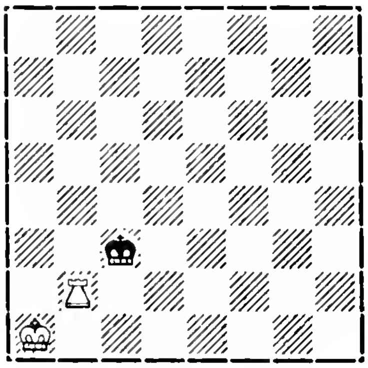
1)



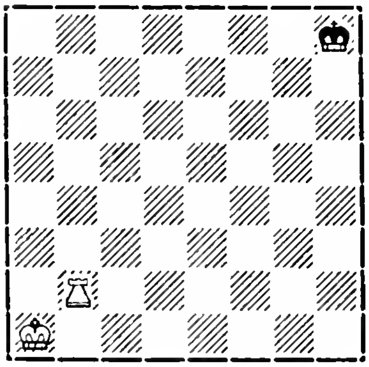
2)



3)

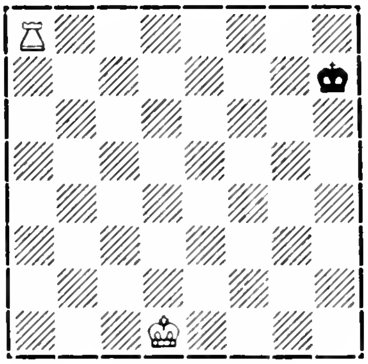


4)

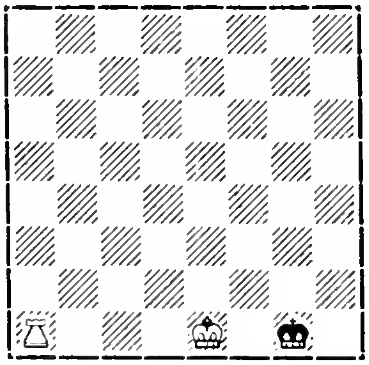


CAMERA was then given two new positions to solve on its own, and was able to achieve checkmate in both cases. These positions were:

1)



2)



Sensing rules which define conditions such as "king is in the corner", "king has two squares in its space", "king is across from king", "king can capture rook", and "there is an empty file or rank between the kings" were input by the trainer in the course of the training session. These rules were needed to enable CAMERA to translate conditions specified by the trainer to control repetitions. For example, the rule "king is in the corner" controls expansion of the goal "reduce the king space", since this goal needs to be repeatedly satisfied only until the king is in the corner.

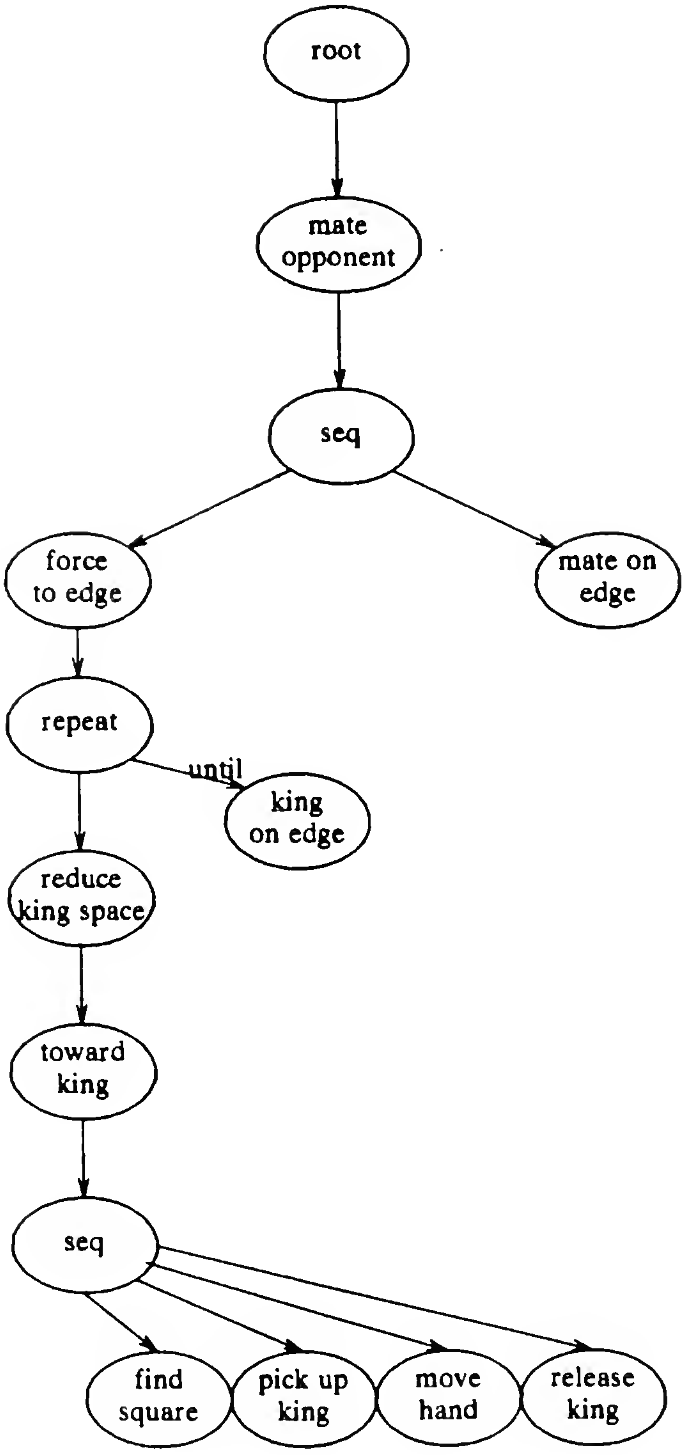
The following are some examples of the above sensing rules:

```
(p king-is-in-the-corner sensing
  (piece ^type king ^x << 1 8 >> ^y << 1 8 >>)
  → )

(p king-has-two-squares-in-its-space sensing
  (production ^pname square-is-in-the-king-space
      ^v1 <x1> ^v2 <y1>)
  (production ^pname square-is-in-the-king-space
      ^v1 <x2> ^v2 <y2>)
 - (production ^pname square-is-in-the-king-space
      ^v1 <x3> ^v2 <y3>)
  → )

(p reduce-the-king-space method
  (goal ^type force-the-king-to-the-edge-of-the-board
      ^status active ^son nil ...)
  →
  (make goal ^type reduce-the-king-space ^status active ...)
  (modify 1 ^son ...) )
```

A portion of the goal tree at one point during the execution of this example is shown on the next page:


root
mate opponent
seq
force to edge
mate on edge
repeat
until
king on edge
reduce king space
toward king
seq
find square
pick up king
move hand
release king

Raw control rules were generated as in the puzzle task. Some of the general control rules produced by DISC are:

```
(p choose-to-force-then-mate control
  (production ^pname force-then-mate ^type method)
  (production ^pname mate ^type method)
 - (production ^pname opponent-king-on-the-edge ^type sensing)
 →
  (call fire-production 1) )
```

which states that if the opponent's king is not on the edge of the board, it is necessary to first force it to the edge of the board;

```
(p choose-to-move-towards-king control
  (production ^pname move-towards-king ^type method)
  (production ^pname move-rook-to-reduce-space ^type method)
  (production ^pname move-rook-to-safety ^type method)
  (production ^pname waste-a-move ^type method)
  (production ^pname king-can-move-closer-to-king ^type sensing)
 →
  (call fire-production 1) )
```

which states that moving toward the opponent's king is desirable whenever possible; and:

```
(p choose-to-move-towards-king control
   (production ^pname move-towards-king ^type method)
   (production ^pname move-rook-to-reduce-space ^type method)
   (production ^pname move-rook-to-safety ^type method)
   (production ^pname waste-a-move ^type method)
   (production ^pname opponent-is-on-the-edge ^type sensing)
   (production ^pname king-is-across-from-king ^type sensing)
 - (production ^pname opponent-is-in-corner ^type sensing)
 →
   (call fire-production 4) )
```

which states that a move should be wasted if the opponent's king is trapped on the edge, and about to be mated, but not in the corner.

7. Conclusions

Production systems have become one of the most widely-used architectures in artificial intelligence research, especially in the field of expert systems. The first major result in this work is the illustration of the usefulness of the conflict set as a means of describing the task environment in the production system paradigm. Since this means of representation represents an extension, not a modification, of the usual production system representation which references only working memory, the production system programmer can use conflict set references to avoid situations in which special tokens or fields of tokens must otherwise be employed. Also, this mechanism permits simple construction of hierarchical production systems, in which a set of rules can control a subordinate set of rules by choosing which of the subordinate rules can fire.

The second major result in this work is the development of the relationship between this representational scheme and the notion of the separation of programming into control and logic, with the logic component consisting of the methods used to solve goals. The use of the conflict set representation strongly parallels this notion, and it was natural to use this representation to implement such a separation. The implemented system utilizes the separation of logic and control to aid in the task of extracting expertise from a trainer, by constructing the two types of information differently. The resulting three-level system corresponds to one of the common models of cognition, which describes cognitive action as consisting of a perceptual level, an intentional level, and a semantic level [Pylyshyn84], which correspond to CAMERA's action level, method level, and control level, respectively. Within this sort of framework, goals represent classes of regular sets of actions. In this particular implementation, goals serve two major functions: they serve as the means of communication of intentional ideas between the trainer and the system; and the use of goal types aids in the generalization process, by adding structure to the control rules.

This task, like all research in machine learning, is concerned with translating a problem description into a machine-executable program. As such, this task requires the ability to represent and construct the three basic control structures of computer programs: sequencing, selection, and repetition. The implementation contains three sets of rules: the action rules, which can be viewed as the assembly language of the task; the method rules, which constitute the context-free component of the system's control, and implement sequencing and repetition; and control rules, the context-sensitive control component, which implement selection. Sensing rules were developed as the mechanism for controlling repetition and for constructing more descriptive control rules. The function of the generalization algorithm can be viewed as that of detecting, or, if necessary, extracting from the trainer, sensing rules to effect selection.

The third accomplishment is the ability of the system to use large rule sets. This is one of the primary difficulties facing the use of production systems in large, complex problems. To solve a task, the system constructs and executes a program in a top-down manner, ignoring productions whose instantiation would be irrelevant to the active goal.

The fourth result of this work is the development of a representational language for generalization. A descriptive language for representing generalizations is constructed which uses conflict set references as its fundamental element. Generalization is then modeled as the process of first searching for relevant conjunctions of attributes, and then describing these conjunctions with the conflict set language. Within this framework, generalization biases occur both as orderings of recognizable patterns and as orderings of potential descriptions.

In the examples which were run, this system for the extraction control and method information was sufficient to enable the system to learn strategies for accomplishing tasks, and to apply those strategies to new situations.

8. Directions for Future Research

This research has explored some of the basic issues relevant to use of the conflict set as a representation for a problem-solver which can extract advice from a human expert and generalize it for effective future use. There are issues which have not been addressed yet.

One of the most important issues which has not been addressed is the automatic construction of heuristics for searching the conjunctions of attributes. Future research will concentrate upon this.

The other significant issue to be addressed is the expansion of the power of the conflict-set language. At present, the language resembles a conflict-set version of VL_1 [Michalski83]. This suggests a possible expansion of the language similar to the development of VL_2 [Michalski83]. This could then be accompanied by implementation of algorithms such as STAR [Michalski83] for the pattern-recognizer.

Also, this entire system is in the form of a demonstration system, which was constructed to test some ideas about knowledge representation in production systems, and in machine learning. As a result, numerous design and implementation decisions were made for simplicity, to permit quick investigation of the system's usefulness. Thus, several aspects of the system exist in their simplest form, and could be replaced with more complex forms, which would undoubtedly result in more sophisticated system behavior. These simple components include:

Goals:

More complex models of goal-interaction (e.g., [Wilensky81, Sacerdoti77, Laird84] etc.) could be implemented within CAMERA. These could allow, for instance, detection and strengthening of multiply-occurring subgoals, or implementation of a scheme of costs and worth of goals.

Selection:

The control rules which select methods are structured so that only one control rule can instantiate at a time. This decision was made to investigate to extent to which discriminatory knowledge can be refined. However, there exist other choices of metarule structure. In particular, voting on the methods is possible [Laird84], as are heuristic methods of eliminating rules and preferring rules [Hayes-Roth83].

Actions:

The current implementation fires action rules whenever a goal exists whose type is identical to the action rule's name. This does not give the system flexibility in deciding whether to fire an action rule, or expand the goal further.

Language:

CAMERA at present uses a very simple language interface with the trainer. Replacement of this component with a more complex translator would allow the extraction of more complex information from the trainer.

Interface:

At present, CAMERA can function only in either of two modes: blindly following the trainer, or completely on its own. The ability to shift modes, and the modeling of third-person interactions, would allow the relaxation of the training/non-training distinction, thus permitting more graceful interaction with trainers.

Completeness:

The present system presumes complete access to information about the task environment. The use of non-monotonic reasoning to provide default values could expand the usefulness of the system.

The entire structure of this system can be viewed as a layered production system, in which the action and goal rules access working memory, and control rules access the action and goal rules, both in their inert form and in instantiated form. The set of control rules can be thought of as being accessed by generalization rules, which use pattern recognition primitives on their left-hand sides. This structure suggests the desirability of a production system architecture which would permit rules to be accessed declaratively, as well as procedurally.

Another issue for future work is that of generalizing the action rules and method rules. The present work has concentrated on generalization of the control rules, although constructive generalization of the action rules, as embodied in the use of the top-level action and sensing rules to describe a situation, occurs. In order to accomplish effective generalization of method and action rules, it is necessary to use a more sophisticated language parser in the interface with the trainer, since the present parser lacks both the notion of transformational grammar and the dictionary necessary to detect identical goals and conditions which may be phrased differently. This is one of the first modifications that will be made to the present program.

This entire work bears a strong connection to automatic programming. Compilation of control knowledge and of data structures into a program would be well worth investigating. In addition, conflict set transformations can be viewed as an underlying representation of goals. For example, the goal "solve the jigsaw puzzle" can be viewed as the goal of causing an instantiation of the production "the puzzle is solved" to enter the conflict set. This may, in turn, be defined as causing all instantiations of the production "piece has no edges joined to the puzzle" to be removed from the conflict set, etc. This approach may prove useful in generalizing method and control rules.

Also, there are several ways in which the speed of the system could be increased. Ideally the system would be able to access rules both procedurally and declaratively. This interpreted system would still be able to remember partial matches of rules, and might associate rules into sets depending upon their structure or the language used to parse them initially. In addition, the following efficiencies would improve the performance of the system:

[1] A new network could be implemented for the **production** elements. This network need not match a **production** element against all rules which contain **production** elements, since at compile time the system can tell exactly which rules may reference a given rule. A network reflecting the structure of dependencies between rules would be much faster than simply inserting a **production** token in working memory, and using the normal OPS5 match.

[2] A separate network could be compiled for the method rules. When a **goal** element is added to working memory, or is updated, it is unnecesary to match it against all the action and sensing rules.

[3] A more efficient pattern matcher could be implemented for the trainer's input. The current one is slow. However, current plans call for the upgrading of the entire parser.

[4] The hashing scheme used by OPS5 for working memory - hashing according to the first atomic symbol - could be modified for **goal** and **production** elements. **Goal** elements could be hashed according to their "type" fields, and **production** elements could be hashed by their "pname" fields. This would help eliminate the long lists of **goal** and **production** elements.

[5] The "rehearse" function, which causes a newly-created rule to be able to instantiate immediately, could be streamlined, particularly if a new hashing scheme were implemented, as described above. In addition, the system as it now stands does not display working memory or the goal tree for the trainer. For the trainer's efficiency, a display interface should be constructed to allow viewing of the goal tree and the task elements, and perhaps also the conflict set.

REFERENCES

[Amarel68]

Amarel, Saul, *On Representations of Reasoning about Actions,* Machine Intelligence **3**, 1968.

[Angluin82]

Angluin, Dana, and Smith, Carl H., *A Survey of Inductive Inference: Theory and Methods* Technical Report 250, Yale University, October, 1982.

[Barnett84]

Barnett, J A, *How Much is Control Knowledge Worth?: A Primitive Example,* Artificial Intelligence **22,** 1984.

[Benjamin83]

Benjamin, D Paul, and Harrison, Malcolm C, *A Production System for Learning Plans from an Expert,* Proceedings of the AAAI Conference, 1983.

[Bennett78]

Bennett, J S, et al, *SACON: A Knowledge-based Consultant for Structural Analysis,* Technical Report, Computer Science Department, Stanford University, September, 1978. Memo HPP-78-23.

[Bennett80]

Bennett, J S, and Goldman, D, *CLOT: A Knowledge-based Consultant for Bleeding Disorders,* Technical Report, Computer Science Department, Stanford University, 1980. Memo HPP-80-7.

[Bennett81]

Bennett, J S, and Hollander, C R, *DART: An Expert System for Computer Fault Diagnosis,* Proc. IJCAI-7, 1981.

[Brazdil78]

Brazdil, P, *Experimental Learning Model,* Proc. Third AISB/GI Conference, 1978, 46-50.

[Buchanan77]

Buchanan, Bruce G, and Mitchell, Tom M, *Model-Directed Learning of Production Rules,* Stanford Computer Science Department Report No. STAN-CS-77-597.

[Davis77]

Davis, R, *Interactive Transfer of Expertise: Acquisition of New Inference Rules,* Proc. IJCAI-77.

[Davis80a]

Davis, R, *Reasoning About Control,* Artificial Intelligence **15**, 1980.

[Davis80b]

Davis, R, *Content-reference: Reasoning About Rules,* Artificial Intelligence **15**, 1980.

[DeJong83]

DeJong, Gerald, *Acquiring Schemata Through Understanding and Generalizing Plans,* Proc. IJCAI-8, 1983.

[Dietterich81]

Dietterich, Thomas G, and Michalski, Ryszard S, *Inductive Learning of*

Structural Descriptions, Artificial Intelligence 16, 1981.

[Fisher36]

Fisher, R A, *The Use of Multiple Measurements in Taxonomic Problems,* Ann. Eugenics, Vol. 7, 1936.

[Forgy81]

Forgy, Charles L, *OPS5 Users' Manual,* Report CMU-CS-81-135, Carnegie-Mellon University, July 1981.

[Georgeff82]

Georgeff, M D, *Procedural Control in Production Systems,* Artificial Intelligence **18**, 1982.

[Georgeff79]

Georgeff, M D, *A Framework for Control in Production Systems,* Proc. IJCAI-6, 1979.

[Hayes-Roth83]

Hayes-Roth, Frederick, Waterman, Donald A, and Lenat, Douglas B, *Building Expert Systems,* Addison-Wesley, 1983.

[Hayes-Roth78]

Hayes-Roth, Frederick, and McDermott, John, *An Interference Matching Technique for Inducing Abstractions,* CACM, Vol.21, No. 5, May 1978.

[Heiser78]

Heiser, J F, Brooks, R E, and Ballard, J P, *Progress Report: A Computerized Psychopharmacology Advisor,* in Proc. 11th Collegium Internationale Neuro-Psychopharmacologicum, Vienna, 1978.

[Hunt75]

Hunt, Earl B., *Artificial Intelligence*, Academic Press, New York, 1975.

[Kibler83]

Kibler, Dennis, and Porter, Bruce, *Perturbation: A Means for Guiding Generalization*, Proc. IJCAI-8, 1983.

[Kline81]

Kline, Paul J, *The Superiority of Relative Criteria in Partial-Matching and Generalization*, Proc. IJCAI-7, 1981.

[Kowalski79]

Kowalski, Robert A, *Algorithm = Logic + Control*, **CACM**, July 1979.

[Kunz78]

Kunz, J C, et al, *A Physiological Rule-based System for Interpreting Pulmonary Function Test Results*, Technical Report, Heuristic Programming Project, Computer Science Department, Stanford University, 1978, HPP-78-19 Working Paper.

[Laird83a]

Laird, John E, and Newell, Allen, *Universal Subgoaling: An Initial Investigation*, Technical Report, Computer Science Department, Carnegie-Mellon University, March, 1983.

[Laird83b]

Laird, John E, and Newell, Allen, *A Universal Weak Method*, Technical Report CMU-CS-83-141, Computer Science Department, Carnegie-Mellon University, June,1983.

[Laird84]

Laird, John E, Rosenbloom, Paul S, and Newell, Allen, *Towards Chunking as a General Learning Mechanism,* Technical Report, Computer Science Department, Carnegie-Mellon University, June,1984.

[Langley83]

Langley, P, *Learning Effective Search Heuristics,* Proc. IJCAI-8, 1983.

[Langley79]

Langley, Pat, Bradshaw, Gary L, and Simon, Herbert A, *BACON.5: The Discovery of Conservation Laws,* Proc. IJCAI-6, 1979.

[Lenat83]

Lenat, D *State of the Art in Machine Learning,* Invited Address, AAAI83 Conference.

[McDermott78]

McDermott, J, and Forgy, C L, *Production System Conflict Resolution Strategies,* in Waterman and F Hayes-Roth, (editors), *Pattern-Directed Inference Systems,* p.177-199, Academic Press, 1978.

[Michalski78]

Michalski, Ryszard S., *Selection of Most Representative Training Examples and Incremental Generalization of VL1 Hypotheses: the Underlying Methodology and the Description of Programs ESEL and AQ11,* Rept. No. 78-867, Department of Computer Science, University of Illinois at Urbana-Champaign, 1978.

[Michalski80]

Michalski, Ryszard S., *Pattern Recognition as Rule-Guided Inductive*

Inference, IEEE Trans. Pattern Anal. Machine Intelligence (1980).

[Michalski83]

Michalski, Ryszard S., *A Theory and Methodology of Inductive Learning*, Artificial Intelligence **20**, 1982.

[Mitchell77]

Mitchell, Tom M, *Version Spaces: A Candidate Elimination Approach to Rule Learning*, Proc. IJCAI **5** (1977), 305-310.

[Mitchell79]

Mitchell, Tom M, Utgoff, Paul E, Nudel, Bernard, and Banerji, Ranan, *Learning Problem-Solving Heuristics Through Practice*, Proc. IJCAI-6, 1979.

[Mitchell82]

Mitchell, Tom M, *Generalization as Search*, Artificial Intelligence **18**, 1982.

[Newell72]

Newell, A, and Simon, H A, *Human Problem Solving*, Prentice-Hall, Englewood Cliffs, 1972.

[Newell73]

Newell, A, *Production Systems: Models of Control Structures*, in *Visual Information Processing*, W. Chase (ed.), Academic Press, 1973.

[Newell77]

Newell, A, *Knowledge Representation Aspects of Production Systems*, Proc. IJCAI-5, 1977.

[Newell80]

Newell, A, *Reasoning, problem solving and decision processes: The problem space as a fundamental category,* in R. Nickerson, Ed., *Attention and Performance VIII,* Erlbaum, Hillsdale, NJ, 1980.

[Newell82a]

Newell, A, *Intellectual Issues in the History of Artificial Intelligence,* Report CMU-CS-82-142, Carnegie-Mellon University, October, 1982.

[Newell82b]

Newell, A, *The Knowledge Level,* Artificial Intelligence **18**, 1982.

[Nilsson65]

Nilsson, N J, *Learning Machines,* McGraw-Hill, New York, 1965.

[Pylyshyn84]

Pylyshyn, Zenon W, *Computation and Cognition,* MIT Press, 1984.

[Reboh80]

Reboh, Rene, *Using a Matcher to Make an Expert Consultation System Behave Intelligently,* Proc. AAAI Conference, 1980.

[Rychener76]

Rychener, Michael D, *Production Systems as a Programming Language for Artificial Intelligence Applications,* Ph. D. thesis, Carnegie_Mellon University, December 1976.

[Sacerdoti77]

Sacerdoti, E D, *A Structure for Plans and Behavior,* Elsevier, New York, 1977.

[Schank77]

Schank, R, and Ableson, R, *Scripts, Plans, Goals, and Behavior,* Elsevier, New York, 1977.

[Sleeman83]

Sleeman, Derek H, *Inferring Student Models for Intelligent Computer-Aided Instruction,* in Machine Learning, An Artificial Intelligence Approach, Michalski, R, Carbonell, J, and Mitchell, T, (editors), 1983.

[Stolfo79]

Stolfo, Salvatore J, and Harrison, Malcolm C, *Automatic Discovery of Heuristics for Nondeterministic Programs,* Proc. IJCAI-6, 1979.

[Stolfo79b]

Stolfo, Salvatore J, and Harrison, Malcolm C, *Automatic Discovery of Heuristics for Nondeterministic Programs,* (full version), Technical Report No.7, Courant Institute, 1979.

[Stolfo79c]

Stolfo, Salvatore J, *Automatic Discovery of Heuristics for Non-deterministic Programs from Sample Execution Traces,* PhD Thesis, New York University, 1979.

[Stolfo82]

Stolfo, Salvatore J, Private Communication.

[Utgoff83]

Utgoff, Paul E, *Adjusting Bias in Concept Learning,* Proc. IJCAI-8, 1983.

```
(p pass-done-up tree
  (<<  goal  root >> ^status active ^son <s> ^type <t>)
  (<<  goal  and  or seq repeat if >> ^status done ^ownid <s>)
 → (write (crlf) <t> is done)
    (modify 1 ^status done) )

;or-done handles 'or' nodes


(p or-done tree
  (or ^status active ^ownid <x>)
  (<<  goal  and  or seq repeat >> ^status done ^father <x>)
 → (modify 1 ^status done) )

;and-seq-done handles 'and' and 'seq' nodes


(p and-seq-done tree
  (<< and seq >> ^status active ^ownid <x>)
  (<<  goal  and  or seq repeat >> ^status done ^father <x>)
 - (<<  goal  and  or seq repeat >> ^status active ^father <x>)
 → (modify 1 ^status done) )
```

```
(p choose-control top-level
  (production ^pname <p> ^type control)
 - (production ^type << tree train interface >> )
 →
  (call fire-production 1) )

(p no-top-levelrules-active top-level
  (root ^status active)
 - (production ^type << tree train interface control >> )
 →
  (write (crlf) I do not know what to do. (crlf) )
  (halt) )

;tree productions


(p problem-done tree
  (root ^status done) → (halt) )


;goal-done passes 'done' back up the goal tree to
;  'root' and 'goal' nodes
```

APPENDIX 2

Domain-independent CAMERA Rules
including Top-level, Tree, and Tree-sensing Rules

```
;top-level rules
(p choose-train top-level
   (production ^pname <p> ^type train)
 - (production ^type tree)
  -->
   (call fire-production 1) )

(p choose-tree top-level
   (production ^pname <p> ^type tree)
  -->
   (call fire-production 1) )

(p choose-interface top-level
   (production ^pname <p> ^type interface)
 - (production ^type << tree train >> )
  -->
   (call fire-production 1) )
```

```
(make eye ^looking-at nothing)
(make hand ^holding nothing)
```

```
;working memory initialization

(make piece ^id 1 ^location in-heap)
(make piece ^id 2 ^location in-heap)
(make piece ^id 3 ^location in-heap)
(make piece ^id 4 ^location in-heap)
(make edge ^side left ^pieceid 1 ^value straight ^color red ^joined no)
(make edge ^side right ^pieceid 1 ^value 1 ^color red ^joined no)
(make edge ^side top ^pieceid 1 ^value straight ^color red ^joined no)
(make edge ^side bottom ^pieceid 1 ^value 4 ^color blue ^joined no)
(make edge ^side left ^pieceid 2 ^value 1 ^color red ^joined no)
(make edge ^side right ^pieceid 2 ^value straight ^color red ^joined no)
(make edge ^side top ^pieceid 2 ^value straight ^color red ^joined no)
(make edge ^side bottom ^pieceid 2 ^value 2 ^color blue ^joined no)
(make edge ^side left ^pieceid 3 ^value straight ^color blue ^joined no)
(make edge ^side right ^pieceid 3 ^value 3 ^color blue ^joined no)
(make edge ^side top ^pieceid 3 ^value 4 ^color blue ^joined no)
(make edge ^side bottom ^pieceid 3 ^value straight ^color blue ^joined no)
(make edge ^side left ^pieceid 4 ^value 3 ^color blue ^joined no)
(make edge ^side right ^pieceid 4 ^value straight ^color blue ^joined no)
(make edge ^side top ^pieceid 4 ^value 2 ^color blue ^joined no)
(make edge ^side bottom ^pieceid 4 ^value straight ^color blue ^joined no)
(make puzzle ^number 0)
(make heap ^number 4)
```

```
(p edge-of-the-piece-matches-an-edge-in-the-puzzle sensing
   (eye ^looking-at <p>)
   (piece ^id <p> ^location { <w> <> in-puzzle })
   (piece ^id <p2> ^location in-puzzle)
   (edge ^pieceid <p> ^value { <v> <> straight } ^joined no)
   (edge ^pieceid <p2> ^value { <v> <> straight } ^joined no)
  → )

(p let-go-of-the-piece-in-the-puzzle action
   (hand ^holding <p>)
   (puzzle ^number <n>)
   (piece ^id <p> ^location { <w> <> in-puzzle })
  →
   (modify 1 ^holding nothing)
   (modify 2 ^number (compute <n> + 1))
   (modify 3 ^location in-puzzle) )

(p the-puzzle-is-not-finished sensing
   (piece ^location { <w> <> in-puzzle })
  → )

(p the-puzzle-is-finished sensing
   (puzzle)
  - (production ^pname the-puzzle-is-not-finished)
  → )
```

```
(p edge-has-attributes sensing
 {(edge ^pieceid <p> ^side <s> ^color <c> ^value <v> ^joined <j>)
                                             <edge> }
 → )

(p join-an-edge-of-the-piece-to-an-edge-in-the-puzzle action
  (hand ^holding <p>)
 {(piece ^id <p> ^location { <w> <> in-puzzle })
                               <piece-in-the-hand> }
  (eye ^looking-at <p>)
 {(piece ^id <p2> ^location in-puzzle) <piece-in-the-puzzle> }
 {(edge ^pieceid <p> ^value { <v> <> straight } ^joined no)
                     <edge-of-the-piece-in-the-hand> }
 {(edge ^pieceid <p2> ^value { <v> <> straight } ^joined no)
                     <edge-of-the-piece-in-the-puzzle> }
 →
  (modify 5 ^joined yes)
  (modify 6 ^joined yes) )
```

```
(p put-the-piece-down-in-the-heap action
  (hand ^holding <p>)
  (piece ^id <p> ^location in-hand)
  (heap ^number <n>)
 →
  (modify 1 ^holding nothing)
  (modify 2 ^location in-heap)
  (modify 3 ^number (compute <n> + 1)) )

(p put-the-piece-down-in-the-puzzle action
  (hand ^holding <p>)
  (piece ^id <p> ^location in-hand)
  (puzzle ^number <n>)
 →
  (modify 1 ^holding nothing)
  (modify 2 ^location in-puzzle)
  (modify 3 ^number (compute <n> + 1)) )

(p piece-has-attributes sensing
 {(piece ^id <p> ^location <w>) <piece> }
 → )

(p heap-is-empty sensing
  (heap ^number 0)
 → )
```

```
(p pick-up-the-piece-in-view action
   (hand ^holding nothing)
   (eye ^looking-at <p>)
   (piece ^id <p> ^location <l>)
   (heap ^number <n>)
  →
   (modify 1 ^holding <p>)
   (modify 3 ^location in-hand)
   (modify 4 ^number (compute <n> - 1)) )

(p looking-at-a-matching-piece sensing
   (eye ^looking-at <p>)
  {(piece ^id <p> ^location { <w> <> in-puzzle })
              <piece-being-looked-at> }
  {(piece ^id <p2> ^location in-puzzle) <piece-in-the-puzzle> }
  {(edge ^pieceid <p> ^value { <e> <> straight })
                      <edge-of-the-piece-being-looked-at> }
  {(edge ^pieceid <p2> ^value { <e> <> straight })
                      <edge-of-the-piece-in-the-puzzle> }
  → )
```

APPENDIX 1

OPS5 Declarations, Action Rules, and Sensing Rules for the Puzzle

```
;declarations of token types


(literalize piece id location)
(literalize edge side pieceid color value joined)
(literalize heap number)
(literalize puzzle number)
(literalize hand holding)
(literalize eye looking-at)

;action productions


(p look-at-a-piece action
   (eye ^looking-at <p>)
  {(piece ^id <p2> ^location <l>) <piece> }
  →
   (modify 1 ^looking-at <p2>) )

(p close-the-eye action
   (eye ^looking-at <p>)
  →
   (modify 1 ^looking-at nothing) )
```

[Yu79]

Yu, V L, et al, *Evaluating the Performance of a Computer-based Consultant,* Computer Programs in Biomedicine 9(1); p.95-102, January, 1979.

[Vere80]

Vere, S A, *Multilevel Counterfactuals for Generalizations of Relational Concepts and Productions,* Artificial Intelligence **14**, 1980.

[Vere78]

Vere, S A, *Inductive Learning of Relational Productions,* in D A Waterman and F Hayes-Roth (Eds.), *Pattern-Directed Inference Systems,* Academic Press, New York, 1978.

[Whitehill80]

Whitehill, Stephen B, *Self-Correcting Generalization,* Proc. AAAI Conference, 1980.

[Wilensky81]

Wilensky, Robert, *Meta-Planning: Representing and Using Knowledge about Planning in Problem-Solving and Natural Language Understanding,* Cognitive Science, 5, 1981.

[Wilkins84]

Wilkins, D E, *Domain-independent Planning: Representation and Plan Generation,* Artificial Intelligence **22**, 1984.

[Winston75]

Winston, Patrick H, *Learning Structural Descriptions from Examples,* in *The Psychology of Computer Vision,* edited by Winston, McGraw-Hill, 1975.

[Wysotzki83]

Wysotzki, Fritz, *Representation and Induction of Infinite Concepts and Recursive Action Sequences,* Proc. IJCAI-8, 1983.

```
;the following rules handle extraneous sons of done 'or' nodes

(p mark-inactive tree
   (or ^status done ^ownid <x>)
   (<< and or seq goal repeat >> ^father <x>
         ^status { <> inactive <> done })
  →
   (modify 2 ^status inactive) )

(p pass-inactive-down tree
   (<< and or seq goal repeat >> ^ownid <x> ^status inactive)
   (<< and or seq goal repeat >> ^father <x>
                        ^status { <> inactive })
  →
   (modify 2 ^status inactive) )

;test-repeat tests for termination of REPEAT goals

(p test-repeat tree
   (repeat ^status active ^ownid <id1>)
   (goal ^status done ^father <id1>)
 → (call test-repeat <id1>) )
```

```
;make-goal-active starts deleting
;  the goal tree under an
;  unsatisfied repeat node

(p make-goal-active tree
   (repeat ^status unsatisfied ^son <id1>)
   (goal  ^status done ^ownid <id1> ^son <s>)
  → (modify 2 ^status active ^son nil)
     (make delete-this ^id <s>)
     (modify 1 ^status active) )

;error-repeat detects the error of a REPEAT without a GOAL
;  directly under it

(p error-repeat tree
   (repeat ^son <s>)
  (<< and or not seq repeat  >> ^ownid <s>)
 → (write (crlf) illegal repeat node)
     (write (crlf) <s>)  (halt) )
```

```
;delete-node & delete-nil implement the deletion of nodes below
;  a REPEAT node

(p delete-node tree
   (delete-this ^id <id1> )
   (<< and or goal repeat seq >> ^ownid <id1> ^son <s>
      ^brother <b> )
  → (remove 1)
      (remove 2)
      (make delete-this ^id <s>)
      (make delete-this ^id <b>) )

(p delete-nil tree
   (delete-this ^id nil)
  → (remove 1) )
```

```
;pass-now-sequence passes 'now' from a done goal to its brother


(p pass-now-sequence tree
   (seq ^status active ^ownid <id1>)
   (<< and or seq goal repeat >> ^status done ^brother <b>
          ^father <id1>)
   (<< and or seq goal repeat >> ^status active ^ownid <b>
          ^when nil ^father <id1>)
  →
   (modify 3 ^when now))

;pass-now-down passes 'now' down the tree except for IF


(p pass-now-down tree
   (<< and or root seq repeat goal >> ^when now ^son <s>
                                       ^status active)
   (<< and or root seq repeat goal >> ^status active ^ownid <s>
                                       ^when nil )
  → (modify 2 ^when now) )
```

```
;pass-now-horizontal passes 'now' among the sons of an AND or OR

(p pass-now-horizontal tree
  (<< and or >> ^ownid <id1> ^status active)
  (<any> ^father <id1> ^status active ^brother <b> ^when now)
  (<any2> ^ownid <b> ^status active ^when nil)
 -->
  (modify 3 ^when now) )

;The next 4 productions ask whether the system will be trained


(p train1 train
  (root ^son nil)
 - (ops-training)
 - (ops-nontraining)
 -->
  (write (crlf) Do you wish to train the system? Type "yes" or "no": )
  (make ops-train ^answer (accept)) )

(p train2 train
  (ops-train ^answer yes)
 -->
  (remove 1)
  (make ops-training) )
```

```
(p train3 train
  (ops-train ^answer no)
 -->
  (remove 1)
  (make ops-nontraining) )

(p train4 train
  (ops-train ^answer { <> yes <> no })
 -->
  (remove 1)
  (write (crlf) Type "yes" or "no" : )
  (make ops-train ^answer (accept)) )

;The next 7 rules implement the interface control


(p interface1 interface
  (<< goal root >> ^status active ^son nil ^when now)
  (ops-training)
 -->
  (call show-rules) )
```

```
;interface2 chooses an existing method rule

 (p interface2 interface
   (ops-training)
   (ops-max-g-rules ^max { <m> <> 0 } )
   (ops-g-rule ^number { <n> > 0 < <m> } )
 -->
   (remove 2 3)
   (call choose-g-rule <n>) )

;interface3 chooses to build a new method rule

(p interface3 interface
   (ops-training)
   (ops-max-g-rules ^max { <m> <> 0 } )
   (ops-g-rule ^number <m>)
 -->
   (remove 2 3)
   (call build-new-rule) )
```

```
;interface4 handles an illegal choice

(p interface4 interface
   (ops-training)
   (ops-max-g-rules ^max { <m> <> 0 } )
   (ops-g-rule ^number { <= 0 } )
 -->
   (remove 3)
   (write (crlf) Type the number of the option you choose: )
   (make ops-g-rule ^number (accept)) )

;interface5 handles an illegal choice

(p interface5 interface
   (ops-training)
   (ops-max-g-rules ^max { <m> <> 0 } )
   (ops-g-rule ^number { > <m> } )
 -->
   (remove 3)
   (write (crlf) Type the number of the option you choose: )
   (make ops-g-rule ^number (accept)) )
```

```
;interface6 builds a new method rule since
; no method rule expands the current goal

(p interface6 interface
   (ops-training)
   (ops-max-g-rules ^max 0)
 -->
   (remove 2)
   (call build-new-rule) )

(p interface7 interface
   (ops-training)
   (ops-max-g-rules ^max { <any> <> 0 })
 - (ops-g-rule ^number <any2>)
 -->
   (write (crlf) Type the number of the option you choose: )
   (make ops-g-rule ^number (accept)) )
```

```
;rule to fire productions

(p fire-production tree
   (production ^pname <p> ^id <id1>)
   (goal ^type <p> ^status active ^when now ^son nil ^ownid <id2>)
 - (goal-param ^goalid <id> ^interpreted nil)
 - (production ^pname instantiation-disagrees-with-goal
        ^v2 <id1> ^v3 <id2>)
  →
   (modify 2 ^status done)
   (call fire-production 1) )

(p instantiation-disagrees-with-goal tree-sensing
   (production ^pname <any> ^type instantiation-disagreeing-with-goal
        ^v2 <id1> ^v4 <id2>)
  → )
```

```
;the following set of rules checks whether variables in a goal
;  and in an instantiation disagree

(p idwg1 instantiation-disagreeing-with-goal
   (production ^pname <p> ^id <id1> ^v1 <v1>)
   (goal ^type <p> ^status active ^son nil ^ownid <id2>
        ^v1 { <> <v1> <> nil })
  -->)

(p idwg2 instantiation-disagreeing-with-goal
   (production ^pname <p> ^id <id1> ^v2 <v2>)
   (goal ^type <p> ^status active ^son nil ^ownid <id2>
        ^v2 { <> <v2> <> nil })
  -->)

              . . .

(p idwg41 instantiation-disagreeing-with-goal
   (production ^pname <p> ^id <id1> ^v41 <v41>)
   (goal ^type <p> ^status active ^son nil ^ownid <id2>
        ^v41 { <> <v41> <> nil })
  -->)
```

```
(p interpret-goal-parameters tree
   (goal ^type <t> ^status active ^when now ^son nil ^ownid <id>)
   (goal-param ^goalid <id> ^ce <ce> ^attr <a> ^value <v>
              ^interpreted nil)
   (ops-var ^pname <t> ^ce <ce> ^attr <a> ^vn <n>)
  -->
   (modify 2 ^interpreted yes)
   (call interpret-param <n> <v> <id>)  )
```

APPENDIX 3

Method and Generalized Control Rules for the Jigsaw Puzzle as Constructed by the system

```
(p solve-a-jigsaw-puzzle method
  (root ^son nil ^brother <v1> ^status active ^ownid <v2> ^when now)
  →
  (make goal ^status active ^ownid (goal 1) ^father (goal 0)
    ^when nil ^type solve-a-jigsaw-puzzle ^brother nil)
  (modify 1 ^son (goal 1)) (goalinc))


(p fire-solve-a-jigsaw-puzzle-1 control
   (production ^pname solve-a-jigsaw-puzzle ^type method ^v1 nil ^v2 nil)
 →
   (call fire-production 1))
```

```
(p start-the-jigsaw-puzzle-then-solve-the-rest-of-the-jigsaw-puzzle method
  (goal ^son nil ^type solve-the-jigsaw-puzzle ^father <v1>
    ^brother <v2> ^status active ^ownid <v3> ^when now) →
  (make seq ^status active ^ownid (goal 1) ^when nil ^father (goal 0)
    ^son (goal 2) ^brother nil)
  (make goal ^status active ^ownid (goal 2) ^father (goal 1)
    ^when nil ^type start-the-jigsaw-puzzle ^brother (goal 3))
  (make goal ^status active ^ownid (goal 3) ^father (goal 1)
    ^when nil ^type solve-the-rest-of-the-jigsaw-puzzle ^brother nil)
  (modify 1 ^son (goal 1)) (goalinc))

(p fire-start-the-jigsaw-puzzle-then-solve-the-rest-of-the-jigsaw-puzzle-2
  control
   (production
    ^pname start-the-jigsaw-puzzle-then-solve-the-rest-of-the-jigsaw-puzzle
    ^type method ^v1 0 ^v2 nil ^v3 1)
 →
   (call fire-production 1))
```

```
(p look-at-a-piece-in-the-heap-then-pick-up-the-piece-in-view-then-put
      -the-first-piece-in-the-puzzle method
   (goal ^son nil ^type start-the-jigsaw-puzzle ^father <v1>
      ^brother <v2> ^status active ^ownid <v3> ^when now) →
   (make seq ^status active ^ownid (goal 1) ^when nil
      ^father (goal 0) ^son (goal 2) ^brother nil)
   (make goal ^status active ^ownid (goal 2) ^father (goal 1)
      ^when nil ^type look-at-a-piece-in-the-heap ^brother (goal 3))
   (make seq ^status active ^ownid (goal 3) ^when nil ^father (goal 1)
      ^son (goal 4) ^brother nil)
   (make goal ^status active ^ownid (goal 4) ^father (goal 3)
      ^when nil ^type pick-up-the-piece-in-view ^brother (goal 5))
   (make goal ^status active ^ownid (goal 5) ^father (goal 3)
      ^when nil ^type put-the-first-piece-in-the-puzzle ^brother nil)
   (modify 1 ^son (goal 1)) (goalinc))

(p fire-look-at-a-piece-in-the-heap-then-pick-up-the-piece-in-view-then
      -put-the-first-piece-in-the-puzzle-3 control
   (production
      ^pname look-at-a-piece-in-the-heap-then-pick-up-the-piece-in-
      view-then-put-the-first-piece-in-the-puzzle ^type method
      ^v1 2 ^v2 4 ^v3 3)
  →
   (call fire-production 1))
```

```
(p repeatedly-put-pieces-in-the-puzzle-until-the-puzzle-is-finished method
   (goal ^son nil ^type solve-the-rest-of-the-jigsaw-puzzle
      ^father <v1> ^brother <v2> ^status active ^ownid <v3> ^when now)
 →
   (make repeat ^status active ^ownid (goal 1) ^father (goal 0)
      ^when nil ^son (goal 2) ^brother nil ^terminate (goal 3))
   (make goal ^status dormant ^ownid (goal 3) ^father (goal 1)
      ^when nil ^type the-puzzle-is-finished ^brother nil)
   (make goal ^status active ^ownid (goal 2) ^father (goal 1)
      ^when nil ^type put-pieces-in-the-puzzle ^brother nil)
   (modify 1 ^son (goal 1)) (goalinc))

(p fire-repeatedly-put-pieces-in-the-puzzle-until-the-puzzle-is-finished-4
  control
   (production ^pname repeatedly-put-pieces-in-the-puzzle-
      until-the-puzzle-is-finished ^type method ^v1 2 ^v2 nil ^v3 4)
 →
   (call fire-production 1))
```

```
(p find-a-piece-in-the-heap-then-put-the-piece-in-the-puzzle method
   (goal ^son nil ^type put-pieces-in-the-puzzle ^father <v1>
      ^brother <v2> ^status active ^ownid <v3> ^when now)
 →
   (make seq ^status active ^ownid (goal 1) ^when nil ^father (goal 0)
      ^son (goal 2) ^brother nil)
   (make goal ^status active ^ownid (goal 2) ^father (goal 1)
      ^when nil ^type find-a-piece-in-the-heap ^brother (goal 3))
   (make goal ^status active ^ownid (goal 3) ^father (goal 1)
      ^when nil ^type put-the-piece-in-the-puzzle ^brother nil)
   (modify 1 ^son (goal 1)) (goalinc))

(p fire-find-a-piece-in-the-heap-then-put-the-piece-in-the-puzzle-5
      control
   (production ^pname find-a-piece-in-the-heap-then-
      put-the-piece-in-the-puzzle ^type method ^v1 10 ^v2 nil ^v3 11)
 →
   (call fire-production 1))
```

```
(p repeatedly-look-at-a-piece-in-the-heap-until-looking-
      at-a-matching-piece method
   (goal ^son nil ^type find-a-piece-in-the-heap ^father <v1>
      ^brother <v2> ^status active ^ownid <v3> ^when now)
  →
   (make repeat ^status active ^ownid (goal 1) ^father (goal 0)
      ^when nil ^son (goal 2) ^brother nil ^terminate (goal 3))
   (make goal ^status dormant ^ownid (goal 3) ^father (goal 1)
      ^when nil ^type looking-at-a-matching-piece ^brother nil)
   (make goal ^status active ^ownid (goal 2) ^father (goal 1)
      ^when nil ^type look-at-a-piece-in-the-heap ^brother nil)
   (modify 1 ^son (goal 1)) (goalinc))

(p fire-repeatedly-look-at-a-piece-in-the-heap-until-looking-
      at-a-matching-piece-6 control
   (production
      ^pname repeatedly-look-at-a-piece-in-the-heap-until-
      looking-at-a-matching-piece ^type method ^v1 13 ^v2 15 ^v3 14)
  →
   (call fire-production 1))
```

```
(p pick-up-the-piece-in-view-then-join-all-the-edges-to-the-
      puzzle-then-let-go-of-the-piece-in-the-puzzle method
   (goal ^son nil ^type put-the-piece-in-the-puzzle ^father <v1>
     ^brother <v2> ^status active ^ownid <v3> ^when now)
   -->
   (make seq ^status active ^ownid (goal 1) ^when nil ^father
      (goal 0) ^son (goal 2) ^brother nil)
   (make goal ^status active ^ownid (goal 2) ^father (goal 1)
     ^when nil ^type pick-up-the-piece-in-view ^brother (goal 3))
   (make seq ^status active ^ownid (goal 3) ^when nil ^father
      (goal 1) ^son (goal 4) ^brother nil)
   (make goal ^status active ^ownid (goal 4) ^father (goal 3)
     ^when nil ^type join-all-the-edges-to-the-puzzle ^brother (goal 5))
   (make goal ^status active ^ownid (goal 5) ^father (goal 3)
     ^when nil ^type let-go-of-the-piece-in-the-puzzle ^brother nil)
   (modify 1 ^son (goal 1)) (goalinc))
```

```
(p fire-pick-up-the-piece-in-view-then-join-all-the-edges-to-the-
      puzzle-then-let-go-of-the-piece-in-the-puzzle-7 control
   (production
      ^pname pick-up-the-piece-in-view-then-join-all-the-edges-to-the-
      puzzle-then-let-go-of-the-piece-in-the-puzzle ^type method ^v1 13
      ^v2 nil ^v3 15)
  -->
   (call fire-production 1))
```

```
(p repeatedly-join-an-edge-of-the-piece-to-an-edge-in-the-puzzle-
      until-not-edge-of-the-piece-matches-an-edge-in-the-puzzle method
   (goal ^son nil ^type join-all-the-edges-to-the-puzzle ^father <v1>
      ^brother <v2> ^status active ^ownid <v3> ^when now)
   →
   (make repeat ^status active ^ownid (goal 1) ^father (goal 0)
      ^when nil ^son (goal 2) ^brother nil ^terminate (goal 3))
   (make not ^status dormant ^ownid (goal 3) ^when nil ^father
      (goal 1) ^son (goal 4) ^brother nil)
   (make goal ^status dormant ^ownid (goal 4) ^father (goal 3)
      ^when nil ^type edge-of-the-piece-matches-an-edge-in-the-puzzle
      ^brother nil)
   (make goal ^status active ^ownid (goal 2) ^father (goal 1)
      ^when nil ^type join-an-edge-of-the-piece-to-an-edge-in-the-puzzle
      ^brother nil)
   (modify 1 ^son (goal 1)) (goalinc))
```

```
(p fire-repeatedly-join-an-edge-of-the-piece-to-an-edge-in-the-puzzle-
        until-not-edge-of-the-piece-matches-an-edge-in-the-puzzle-8 control
    (production
        ^pname repeatedly-join-an-edge-of-the-piece-to-an-edge-in-
        the-puzzle-until-not-edge-of-the-piece-matches-an-edge-in-the-puzzle
        ^type method ^v1 21 ^v2 23 ^v3 22)
  -->
    (call fire-production 1))

(p fire-find-a-piece-in-the-heap-then-put-the-piece-in-the-puzzle-9 control
    (production
        ^pname find-a-piece-in-the-heap-then-put-the-piece-in-the-puzzle
        ^type method ^v1 10 ^v2 nil ^v3 11)
  -->
    (call fire-production 1))

(p fire-repeatedly-look-at-a-piece-in-the-heap-until-looking-at-a-
        matching-piece-10 control
    (production
        ^pname repeatedly-look-at-a-piece-in-the-heap-until-looking-
        at-a-matching-piece ^type method ^v1 28 ^v2 30 ^v3 29)
  -->
    (call fire-production 1))
```

```
(p fire-pick-up-the-piece-in-view-then-join-all-the-edges-to-the-
      puzzle-then-let-go-of-the-piece-in-the-puzzle-11 control
   (production
      ^pname pick-up-the-piece-in-view-then-join-all-the-edges-to-the-
      puzzle-then-let-go-of-the-piece-in-the-puzzle ^type method ^v1 28
      ^v2 nil ^v3 30)
  →
   (call fire-production 1))

(p fire-repeatedly-join-an-edge-of-the-piece-to-an-edge-in-the-puzzle-
      until-not-edge-of-the-piece-matches-an-edge-in-the-puzzle-12 control
   (production
      ^pname repeatedly-join-an-edge-of-the-piece-to-an-edge-in-the-
      puzzle-until-not-edge-of-the-piece-matches-an-edge-in-the-puzzle
      ^type method ^v1 36 ^v2 38 ^v3 37)
  →
   (call fire-production 1))

(p fire-find-a-piece-in-the-heap-then-put-the-piece-in-the-puzzle-13 control
   (production
      ^pname find-a-piece-in-the-heap-then-put-the-piece-in-the-puzzle
      ^type method ^v1 10 ^v2 nil ^v3 11)
  →
   (call fire-production 1))
```

```
(p fire-repeatedly-look-at-a-piece-in-the-heap-until-looking-at-a-
      matching-piece-14 control
   (production
      ^pname repeatedly-look-at-a-piece-in-the-heap-until-looking-at-
      a-matching-piece ^type method ^v1 43 ^v2 45 ^v3 44)
  -->
   (call fire-production 1))

(p fire-pick-up-the-piece-in-view-then-join-all-the-edges-to-the-puzzle-
      then-let-go-of-the-piece-in-the-puzzle-15 control
   (production
      ^pname pick-up-the-piece-in-view-then-join-all-the-edges-to-the-
      puzzle-then-let-go-of-the-piece-in-the-puzzle ^type method ^v1 43
      ^v2 nil ^v3 45)
  -->
   (call fire-production 1))
```

```
(p fire-repeatedly-join-an-edge-of-the-piece-to-an-edge-in-the-puzzle-
      until-not-edge-of-the-piece-matches-an-edge-in-the-puzzle-16 control
   (production
      ^pname repeatedly-join-an-edge-of-the-piece-to-an-edge-in-the-
      puzzle-until-not-edge-of-the-piece-matches-an-edge-in-the-puzzle
      ^type method ^v1 51 ^v2 53 ^v3 52)
  -->
   (call fire-production 1))
```

APPENDIX 4

A Full Implementation of Sleeman's System for Students' Algebra Skill

Rules are presented in stylized format,
without goal tree pointers, for clarity

```
(p EVAL action
  (lhs M op N rhs) → (lhs (evaluate M op N)) )

(p SOLVE action
  (SHD M * X = N) → (SHD X = N/M) )

(p TORHS action
  (lhs ± term lhs2 = rhs) → (lhs lhs2 = rhs ∓ term) )

(p COLL action
  (lhs M*X ± N*X rhs) → (lhs < M ± N > * X rhs) )

(p DIST action
  (lhs M * < N*X ± P > rhs) → (lhs M*N*X ± M*P) )

(p BRACE action
  (lhs < M > rhs) → (lhs M rhs) )

(p TOLHS action
  (lhs = rhs ± term rhs2) → (lhs ∓ term = rhs rhs2) )
```

METHOD RULES

```
(p solve-the-equation method
  (root)
 -->
  (make goal ^type solve-the-equation) )

(p move-terms-then-evaluate method
  (goal ^type solve-the-equation)
 -->
  (make seq)
  (make goal ^type multiply-out-all-terms)
  (make goal ^type move-all-terms-without-X-to-rhs)
  (make goal ^type move-all-terms-with-X-to-lhs)
  (make goal ^type collect-all-terms-on-lhs)
  (make goal ^type evaluate-everything)
  (make goal ^type solve)
  (make goal ^type evaluate-the-division) )
```

```
(p multiply-out-all-terms-and-remove-braces method
   (goal ^type multiply-out-all-terms)
  -->
;repeatedly DIST until no longer possible, then
; BRACE until no longer possible
   (make seq)
   (make repeat)
   (make goal ^type DIST)
   (make until)
   (make not)
   (make goal ^type DIST)
   (make repeat)
   (make goal ^type BRACE)
   (make until)
   (make not)
   (make goal ^type BRACE) )
```

```
(p repeatedly-move-a-term-without-X-to-rhs method
   (goal ^type move-all-terms-without-X-to-rhs)
  →
   (make repeat)
   (make goal ^type TORHS)
   (make until)
   (make not)
   (make goal ^type term-without-X-on-lhs) )

(p term-without-X-on-lhs sensing
   (lhs ± term ± lhs2 = rhs)
   (neq (substr term n n) X)
  → )

(p repeatedly-move-a-term-with-X-to-lhs method
   (goal ^type move-all-terms-with-X-to-lhs)
  →
   (make repeat)
   (make goal ^type TOLHS)
   (make until)
   (make not)
   (make goal ^type term-with-X-on-rhs) )
```

```
(p term-with-X-on-rhs sensing
   (lhs = rhs ± term1 X term2 ± rhs2)
  → )

(p collect-all-terms method
;repeatedly COLL until no longer possible to COLL
   (goal ^type collect-all-terms-on-lhs)
  →
   (make repeat)
   (make goal ^type COLL)
   (make until)
   (make not)
   (make goal ^type COLL) )

(p evaluate-all-terms method
;repeatedly EVAL until no longer possible to EVAL
   (goal ^type evaluate-everything)
  →
   (make repeat)
   (make goal ^type EVAL)
   (make until)
   (make not)
   (make goal ^type EVAL) )
```

```
(p evaluate-the-division method
  (goal ^type evaluate-the-division)
 →
  (make goal ^type EVAL) )
```

CONTROL RULES

```
(p choose-to-solve-the-equation control
  (production ^pname solve-the-equation)
 →
  (call fire-production 1) )

(p choose-to-move-terms-then-evaluate control
  (production ^pname move-terms-then-evaluate)
 →
  (call fire-production 1) )

(p choose-to-multiply-out-all-terms-and-remove-braces control
  (production ^pname multiply-out-all-terms-and-remove-braces)
 →
  (call fire-production 1) )
```

```
(p choose-to-repeatedly-move-a-term-without-X-to-rhs control
   (production ^pname repeatedly-move-a-term-without-X-to-rhs)
  →
   (call fire-production 1) )

(p choose-to-repeatedly-move-a-term-with-X-to-lhs control
   (production ^pname repeatedly-move-a-term-with-X-to-lhs)
  →
   (call fire-production 1) )

(p choose-to-collect-all-terms control
   (production ^pname collect-all-terms)
  →
   (call fire-production 1) )

(p choose-to-evaluate-all-terms control
   (production ^pname evaluate-all-terms)
  →
   (call fire-production 1) )

(p choose-to-evaluate-the-division control
   (production ^pname evaluate-the-division)
  →
   (call fire-production 1) )
```

```
(p choose-TORHS control
   (production ^pname TORHS ^term <t>)
   (production ^pname term-without-X-on-lhs ^term <t>)
  →
   (call fire-production 1) )

(p choose-TOLHS control
   (production ^pname TOLHS ^term <t>)
   (production ^pname term-with-X-on-rhs ^term <t>)
  →
   (call fire-production 1) )
```

NYU COMPSCI TR-227 c.1
Benjamin, Paul
Extraction and
generalization of expert
advice.

NYU COMPSCI TR-227 c.1
Benjamin, Paul
Extraction and
generalization of expert
advice.